IBM RT PC Advanced Interactive Executive Operating System

VS Language/Operating System Interface Library

Programming Family

IBM
Personal
Computer
Software

SH23-0131

VS Language/Operating System Interface Library

Programming Family

IBM
Personal
Computer
Software

First Edition (March 1987)

The information in this manual applies to of IBM RT PC VS Pascal and of IBM RT PC VS FORTRAN for use with Release 2.1 of the AIX Operating System, and it applies to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. IBM may use or distribute, in any way it believes appropriate and without incurring any obligation to the sender, whatever information it receives in this manner.

Preface

This manual contains information about the library of system calls available with IBM RT PC[1] VS Pascal and IBM RT PC VS FORTRAN as implemented for use with the AIX[2] Operating System.

Contents

Chapter 1—Introduction to the RT PC VS Interface Library—is an overview of the IBM RT PC Interface Library of system calls.

Chapter 2—Process Control—describes system calls that control the creation, operation, and stopping of processes.

Chapter 3—Process Identification—describes system calls that get and set the identifiers and limits of a process.

Chapter 4—Process Tracking—describes system calls that monitor the operation of a process or group of processes.

Chapter 5—Input and Output—describes system calls that invoke the basic input-output operations for the system.

Chapter 6—File Maintenance—describes system calls that invoke a variety of operations on files, including the changing of access permissions, the creation of directories, and the mounting of file systems.

Chapter 7—Signals—describes system calls that interrupt a process and cause it to perform some specified operation.

---

1 RT PC is a trademark of IBM Corporation.

2 AIX is a trademark of IBM Corporation.

Chapter 8—Semaphores—describes system calls whose functions are essentially the same as those of signal calls but represent a more versatile method of interprocess communication.

Chapter 9—Messages—describes system calls that provide a general method for passing information between two processes.

Chapter 10—Shared Memory—describes system calls that reserve an area of memory that can be accessed by cooperating processes, thereby making possible interprocess exchanges of large amounts of data.

Chapter 11—System Utilities—describes a group of miscellaneous routines that have to do with the operating system and the system clock.

Appendix A—Error Codes and Messages—lists the various error codes and messages associated with the AIX Operating System calls.

Appendix B—Pascal and FORTRAN Constant Definitions—lists the constant definitions that are require for Pascal and FORTRAN calling sequences.

Appendix C—Pascal Type Declarations—lists the type declarations that are required for Pascal calling sequences.

Appendix D—Pascal Procedure and Function Declarations—lists procedure and function declarations that are required for Pascal calling sequences.

Appendix E—The ftok System Subroutine—describes a system subroutine that is used in conjunction with certain of the system calls described in this manual.

Appendix F—The perror System Subroutine—describes a system subroutine that is used to write a message that explains an error.

Related Publications

You may want to refer to the following IBM RT PC publications for additional information:

- *VS Pascal User's Guide*, SH23-0127, describes the procedures for compiling and running RT PC VS Pascal programs under the AIX Operating System.
- *VS Pascal Reference Manual*, SH23-0128, describes the Pascal programming language as implemented on the IBM RT PC.
- *VS FORTRAN User's Guide*, SH23-0129, describes the procedures for compiling and running RT PC VS FORTRAN programs under the AIX Operating System.
- *VS FORTRAN Reference Manual*, SH23-0130, describes the FORTRAN 77 programming language as implemented on the RT PC.
- *Concepts*, GC23-0784, gives an overview of the RT PC hardware, the AIX Operating System, and supporting publications.
- *Installing and Customizing the AIX Operating System*, SV21-8001, provides step-by-step instructions for installing and customizing the AIX Operating System, including instructions for adding devices to and deleting them from the system and for defining device characteristics. This book also explains how to create, delete, and change AIX and non-AIX minidisks.
- *Messages Reference*, SV21-8002, lists messages displayed by the RT PC and explains how to respond to the messages.
- *Usability Services Guide* and *Usability Services Reference*, SV21-8003, show how to create and print text files, work with directories, start application programs, and do other basic tasks.
- *Using and Managing the AIX Operating System*, SV21-8004, contains information on using AIX Operating System commands, working with the file system, developing shell procedures, and performing such

system-management tasks as creating and mounting file systems, backing up the system, and repairing file-system damage.

- *AIX Operating System Commands Reference*, SV21-8005, lists and describes the AIX Operating System commands.
- *C Language Guide and Reference*, SV21-8008, provides information for writing, compiling, and running C-language programs.
- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer would use to write application programs. This book also provides information about the AIX Operating System file system, special files, miscellaneous files, and the writing of device drivers.
- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about the use of operating system tools to develop, compile, and debug programs.
- *AIX Operating System DOS Services Reference*, SV21-8012, provides step-by-step information for using the AIX Operating System shell. In addition, this book describes the DOS system services.
- *User Setup Guide*, SV21-8020, provides instructions for setting up and connecting devices to system units. It also gives procedures for installing the AIX Operating System and for testing the setup.
- *Guide to Operations*, SV21-8021, describes system units, displays, console keyboard, and other devices that can be attached to the RT PC. This guide also includes procedures for operating the hardware and for moving system units.
- *Problem Determination Guide*, SV21-8022, provides instructions for running diagnostic routines to locate and identify hardware problems, as well as problem-determination procedures for software.

Contents

Chapter 1. Introduction to the RT PC VS Interface Library

What It Is

The RT PC VS Interface Library is an application-program interface that provides access to the system calls of the AIX Operating System from programs written either in RT PC VS Pascal or in RT PC VS FORTRAN. These system calls, which are a part of the AIX Operating System, invoke a variety of system routines, whose functions include file maintenance, input and output, and interprocess[1] communication.

What You Need

- The AIX Operating System installed on your RT PC
- RT PC VS Pascal or RT PC VS FORTRAN installed according to the directions given in the Program Directory that accompanied the language.

---

1 As used in this manual, the term *process* refers to a program running under the AIX Operating System, together with the environment it runs in.

What It Does

The RT PC VS Interface Library makes it easy to use the AIX system calls directly from programs written in RT PC VS Pascal or RT PC VS FORTRAN by changing the calls's associated data structures, naming conventions, and data types to conform to those required by the system. The Interface Library takes care of many of the details of interfacing to the actual system calls without the need for C-language or assembly-language "wrappers."

How This Manual is Organized

Chapters 2 through 11 divide the AIX system calls into the following groups:

- process control
- process identification
- process tracking
- input and output
- file maintenance
- signals
- semaphores
- messages
- shared memory
- system utilities

In each chapter, a brief introduction is followed by subsections, listed in alphabetical order, describing the individual calls. Each subsection describes the function of the system routine being called, the syntax of the call, its parameters, any return values, and examples of the call from both RT PC VS Pascal and RT PC VS FORTRAN. Where appropriate, additional information is provided in various "Notes."

The calls, listed by group, are as follows:

Process Control

BRK, SBRK (change data-segment space allocation)

EXECL, EXECLE, EXECLP (execute a program)

EXECV, EXECVE, EXECVP (execute a program)

EXIT, EEXIT (terminate a process)

FORK (create a new process)

NICE (set a process priority)

PIPE (create an interprocess channel)

PLOCK (lock or unlock a process, text, or data)

WAIT (delay the calling process)

Process Identification

GETGRP (get a group access list)

GETUID, GETGID, GTEUID, GTEGID (get a user or a group identifier)

GTPGRP, GETPID, GTPPID (get a process-group or a process identifier)

SETGRP (set a group access list)

SETUID, SETGID (set user or group identifiers)

STPGRP (set a process group ID)

ULIMIT (get and set process limits)

USRINF (get and set user information)

Process Tracking

ACCT (turn accounting process on or off)

PROFIL (generate a time profile)

PTRACE (trace the execution of a child process)

TIMES (get the processing times)

Input-Output

ACCESS (check file-access permissions)

CLOSE (close a file)

CREAT (create a new file)

DUP (duplicate an open-file descriptor)

FCLEAR (clear space in a file)

FSYNC (write to permanent storage)

IOCTL (control the input and output of a device)

LOCKF (lock or unlock a region of a file)

LSEEK (set a read or write pointer)

OPEN (open a file for reading or writing)

READ, READX (read from a file)

WRITE, WRITEX (write to a file)

File Maintenance

CHMOD (change file-access permissions)

CHOWN (change file ownership)

CHROOT (change a root directory)

FCNTL (control an open-file descriptor)

FTRUNC (truncate a file)

LINK (link to a file)

MKNOD (create a directory or a special file)

MOUNT, UMOUNT (mount or unmount a file system)

STAT (return the status of a file)

SYNC (update a file system)

UMASK (set and get a file-creation-mode mask)

UNLINK (delete a directory entry)

USTAT (get file-system information)

UTIME (set the file times)

Signals

ALARM (schedule an alarm signal)

KILL (send a signal to a process)

PAUSE (wait for a signal)

SIGBLK (block one or more signals)

SIGMSK (set the signal mask of the current process)

SIGNAL (specify the process response to a signal)

SIGPAS (release a blocked signal and wait for an interrupt)

SIGSTK (define an alternate stack)

SIGVEC (select signal-handling facilities)

Note: The **SIGSTK** and **SIGVEC** calls are not available in FORTRAN.

Semaphores

SEMCTL (invoke semaphore-control operations)

SEMGET (get or create a semaphore-set identifier)

SEMOP (perform semaphore operations)

Messages

MSGCTL (invoke message-control operations)

MSGGET (get or create a message queue)

MSGRV, MSGXRV (read and store a message)

MSGSND (send a message to a queue)

Shared Memory

SHMAT (attach a shared-memory segment or mapped file)

SHMCTL (invoke shared-memory-control operations)

SHMDT (detach a shared-memory or mapped-file segment)

SHMGET (get a shared-memory-segment identifier)

System Utilities

CHDIR (change the default directory)

REBOOT (restart the operating system)

STIME (set the system clock)

TIME (get the system time)

UNAME, UNAMEX (get the name of the current operating system)

The ftok System Subroutine

The RT PC VS Interface Library provides access to AIX Operating System calls. An exception is **ftok**, an AIX Operating System subroutine that is often used by Pascal procedures and FORTRAN subroutines of the kind shown in the program examples elsewhere in this manual. For your convenience, therefore, a description of **ftok** is given in Appendix E.

Using the RT PC VS Interface Library with RT PC VS Pascal

Before you can use the Interface Library with an RT PC VS Pascal program, you must do two things:

1. Declare the constants, data types, and external functions that will be used by the program, so that it can be compiled.

   For your convenience, these declarations are provided in include files (see Appendixes). The type declarations include those for the parameters and return values that appear in the descriptions of the calls. For purposes of illustration, predefined constants, types, and functions listed in the include files are also used in the programming examples.

2. Link the Interface Library to the program, using the **cc** utility.

Once these requirements are satisfied, you can use any number of AIX system calls in your program. For information concerning these calls, see "Related Publications" on page 1-12.

Declarations

The Interface Library provides three files that can be used for making Pascal declarations:

1. constants:

   /usr/include/ailpconsts.inc

2. data types:

   /usr/include/ailtypes.inc

3. external functions:

   /usr/include/aildefs.inc

To include any of these files in an RT PC VS Pascal program, use the %include compiler directive (see *VS Pascal User's Guide*; for the contents

of the include files, see the Appendixes). The program listed here shows how these files are used.

```
program aildemo;

const
  %include /usr/include/ailpconsts.inc

type
  %include /usr/include/ailtypes.inc
  usrary = packed array[1..INFSIZ] of char;
  usrptr = @usrary;

%include /usr/include/aildefs.inc

function p_usrinf (cmd : integer; buf : usrptr;
                   count : integer) : integer; external;

procedure call1;
var
  red : unam;
  blue : integer;

begin
  blue := p_uname (red);
  writeln (red.sysname)
end;

procedure call2;
var
  blue, red : integer;
  yellow : usrptr;

begin
   new (yellow);
  blue := p_usrinf (GETINF, yellow, INFSIZ);
  for red := 1 to blue do
    write (yellow@[red]);
  writeln
end;

begin
  call1;
  call2
end.
```

Linkage

To link the object code of the Interface Library to your program, it is necessary to add the file /usr/lib/libvspil.a to the beginning of the list of files being linked by the **cc** utility. This utility is normally used in the last step of the compiling and linking phase. For example, after obtaining an '.o' file from the program progname.pas, you would use the following (typed as a single command):

```
cc -o progname progname.o -lm /usr/lib/libvspil.a
                              /usr/lib/libvssys.a
```

Using the RT PC VS Interface Library with RT PC VS FORTRAN

Before you can use the Interface Library with an RT PC VS FORTRAN program, you must do two things:

1. First, declare the constants that will be used by the program, so that it can be compiled.

2. Second, link the Interface Library to the program, using the **cc** utility.

Once these requirements are satisfied, you can use any number of AIX system calls in your program. For information concerning these calls, see "Related Publications" on page 1-12.

Declarations

The Interface Library provides one file that can be used for making FORTRAN declarations:

/usr/include/ailfconsts.inc

To include this file in your program, use the INCLUDE compiler directive (see *VS FORTRAN User's Guide*; for a description of the contents of the file, see Appendix B).

The following program illustrates how this file is used:

```
PROGRAM AILDEMO
CALL FIRST
CALL SECOND
END

SUBROUTINE FIRST
CHARACTER*9 RED(5)
INTEGER BLUE, UNAME
BLUE = UNAME (RED)
PRINT *, RED(1)
END

SUBROUTINE SECOND
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER RED, BLUE, USRINF
CHARACTER*INFSIZ YELLOW
BLUE = USRINF (GETINF, YELLOW, INFSIZ)
WRITE  *, YELLOW
END
```

Linkage

To link the object code of the Interface Library to your program, it is necessary to add the file /usr/lib/libvsfil.a to the beginning of the list of files being linked by the **cc** utility. This utility is normally used in the last step of the compiling and linking phase. For example, after obtaining an '.o' file from the program progname.f, you would use the following (typed as a single command):

```
cc -o progname progname.o -lm /usr/lib/libvsfil.a
                              /usr/lib/libvsfor.a
                              /usr/lib/libvssys.a
```

Return Values, Error Codes, and Error Messages

Most of the AIX system calls from Pascal and FORTRAN return a value. See the individual system-call descriptions for details regarding these values.

A return value of -1 indicates that an error has occurred. When a system call generates an error, an error code is set in the external variable *errno*. Two routines are available for retrieving this value:

1. A call to the p__ercode function in Pascal or the ERCODE subroutine in FORTRAN returns the value of the external variable *errno*.

2. A call to **perror** system subroutine prints out an error message (for a description of **perror**, see Appendix F).

Related Publications

A full list of the publications that users of the Interface Library might wish to consult follows the Preface. However, three of these will be of particular interest to the programmer:

- *AIX Operating System Commands Reference*, SV21-8005, describes the AIX Operating System commands.

- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer would use to write application programs. It also provides information about the AIX Operating System file system, special and miscellaneous files, and the writing of device drivers.

- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about the use of operating system tools to develop, compile, and debug programs.

Note: In some instances, an AIX Operating System call has a name that is slightly different from that used by the Interface Library. Where such a difference exists, it is identified in this manual, under **Notes**, at the end of the section describing the call.

Chapter 2. Process Control

The following system calls can be used to control the creation, operation, and stopping of processes.

Call	Function
BRK, SBRK	change data-segment space allocation.
EXEC	is a group of six system calls to routines that run a new program in the currently running process.
EXIT, EEXIT	terminate a process.
FORK	creates a new ("child") process by copying the calling ("parent") process.
NICE	sets the priority of a process.
PIPE	creates an interprocess channel.
PLOCK	locks a process, text, or data in memory.
WAIT	delays the calling process.

BRK, SBRK
— change data-segment space allocation

Description

The **BRK** and **SBRK** system calls dynamically change the amount of space allocated to the data segment of the calling process.

The **BRK** system call sets the breakpoint value to that specified in the call and changes the space allocation accordingly.

The **SBRK** system call adds to the breakpoint value the number of bytes specified in the call and changes the space allocation accordingly.

Syntax

Pascal

```
p_brk (endds);

p_sbrk (incr);
```

FORTRAN

```
BRK (ENDDS)

SBRK (INCR)
```

Parameters

endds
is used only with the **BRK** call. It specifies the new breakpoint that is to be set.

- In Pascal, *endds* is of type integer.
- In FORTRAN, *endds* is of type INTEGER.

incr

is used only with the **SBRK** call. It specifies the number of bytes to be added to or subtracted from the space allocated to the program data segment.

- In Pascal, *incr* is of type integer.
- In FORTRAN, *incr* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the **BRK** call.

The previous break value is returned upon successful completion of the **SBRK** call.

The value -1 is returned and an error code set in *errno* if either call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here issue an **SBRK** system call to add 1000 bytes to the data segment of the calling program. The return value is in the variable "blue".

Pascal

```
procedure sbrk1;

type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := 1000;
  blue := p_sbrk (red);
  writeln (blue)
end;
```

FORTRAN

```
SUBROUTINE SBRK1
INTEGER RED, BLUE
RED = 1000
BLUE = SBRK (RED)
PRINT *, BLUE
END
```

EXECL, EXECLE, EXECLP
— execute a program

Description

The **EXEC** system call, in all its forms, executes a new program in the calling process. The call does not create a new process but overlays the current program with a new one.

The three **EXEC** calls described in this section pass a maximum of four arguments to a specified executable file. This restriction on the number of arguments is what distinguishes these three system calls from those described in the next section.

The **EXECLE** call differs from the other two in having an *envp* parameter.

The **EXECLP** call is issued with the same arguments as **EXECL**, but it duplicates the shell actions in searching for an executable file in a list of directories.

Syntax

Pascal

```
p_execl (path, arg0, arg1, arg2, arg3);

p_execle (path, arg0, arg1, arg2, arg3, envp);

p_execlp (filenm, arg0, arg1, arg2, arg3);
```

---

* The **EXECV**, **EXECVE**, and **EXECVP** calls are described in the next section (page 2-9).

FORTRAN

```
EXECL (PATH, ARG0, ARG1, ARG2, ARG3)

EXECLE (PATH, ARG0, ARG1, ARG2, ARG3, ENVP)

EXECLP (FILENM, ARG0, ARG1, ARG2, ARG3)
```

Parameters

path
is the explicit path (location) of the file to be executed. This parameter is used in the **EXECL** and **EXECLE** calls.

- In Pascal, *path* is of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be a blank space.

filenm
is the name of the file to be executed. This parameter is used in the **EXECLP** call, which will search for the specified file only in the current and default directories.

- In Pascal, *filenm* is of type st80.
- In FORTRAN, *filenm* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be a blank space.

arg0, *arg1*, *arg2*, and *arg3*
are string variables or constants. They hold the arguments to be passed to the file specified by *filenm* or *path*. The value of *arg0* must be *filenm* or the last attribute of *path*.

- In Pascal, each *arg* is of type st80. If fewer than four arguments are required, the remaining strings must be nil strings.
- In FORTRAN, each *arg* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be

a blank space. If fewer than four arguments are required, the remaining strings must each contain one, and only one, blank.

envp

is a parameter used only in **EXECLE** (and **EXECVE**; see next section). It is an 80-element array that holds the attributes of the execution environment of the calling process. (Each element is an 80-byte character string.)

- In Pascal, *envp* is a variable of type pasargv. The terminating string in the array must be a nil string.
- In FORTRAN, *envp* is an array of strings of type CHARACTER*80. The terminating character of a string must be a blank space. The terminating string in the array must contain one, and only one, blank.

  Note: For details of this parameter, see the **sh** command in *AIX Operating System Commands Reference*.

Return Values

There is no return value from a successful **EXEC** call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the next page call the **EXECL** system routine, which prints the current date.

Pascal

```
procedure execl1;

type
  %include /usr/include/ailtypes.inc
var
  merlin : integer;
  path, arg0, arg1, arg2, arg3 : st80;

%include /usr/include/aildefs.inc

begin
  path := '/bin/sh';
  arg0 := 'sh';
  arg1 := '-c';
  arg2 := 'date';
  arg3 := '';
  merlin := p_execl (path, arg0, arg1, arg2, arg3)
end;
```

FORTRAN

```
      SUBROUTINE EXECL1
      INTEGER MERLIN
      CHARACTER*80 PATH, ARG0, ARG1, ARG2, ARG3
      PATH = '/bin/sh '
      ARG0 = 'sh '
      ARG1 = '-c '
      ARG2 = 'date '
      ARG3 = ' '
      MERLIN = EXECL (PATH, ARG0, ARG1, ARG2, ARG3)
      END
```

EXECV, EXECVE, EXECVP
— execute a program

Description

The three **EXEC** system calls described in this section can pass a maximum of 80 arguments to a specified executable file (in contrast to the maximum of four arguments that can be passed by the **EXEC** routines described in the preceding section).

The **EXECVE** call differs from the other two in having an *envp* parameter.

The **EXECVP** call is issued with the same arguments as **EXECV**, but it duplicates the shell actions in searching for an executable file in a list of directories.

Syntax

Pascal

```
p_execv (path, args);

p_execve (path, args, envp);

p_execvp (filenm, args);
```

FORTRAN

```
EXECV (PATH, ARGS)

EXECVE (PATH, ARGS, ENVP)

EXECVP (FILENM, ARGS)
```

Parameters

path
is the explicit path (location) of the file to be executed. This parameter is used in the **EXECV** and **EXECVE** calls.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be a blank space.

filenm
is the name of the file to be loaded and executed. This parameter is used in the **EXECVP** call, which will search for the specified file only in the current and default directories.

- In Pascal, *filenm* is a string variable or constant of type st80.
- In FORTRAN, *filenm* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be a blank space.

args
is an array of strings. It holds any arguments to be passed to the file specified by *filenm* or *path*. The first element of the array should be *filenm* or the last attribute of *path*.

- In Pascal, *args* is a variable of type pasargv declared in the types file. The terminating string must be a nil string.
- In FORTRAN, *args* is a string variable or constant of type CHARACTER*80. The terminating character of a string must be a blank space. The terminating string must contain one, and only one, blank.

envp
is a parameter used only in **EXECVE** (and **EXECLE**, described in the preceding section). It is an 80-element array that holds the attributes of the execution environment of the calling process. (Each element is an 80-byte character string.)

- In Pascal, *envp* is a variable of type pasargv, declared in the types file. The terminating string in the array must be a nil string.
- In FORTRAN, *envp* is an array of strings of type CHARACTER*80. The terminating character of a string must be a blank space. The terminating string in the array must contain one, and only one, blank.
- For details of this parameter, see the description of the **sh** command in *AIX Operating System Commands Reference*.

Return Values

There is no return value from a successful **EXEC** call. The value -1 is returned and an error code set in *errno* if the call fails.

If **EXECVP** is called to execute a shell command file and it is impossible to execute that file, the values of args[0] and args[1] are modified before the return.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **EXECV** system routine, which will produce a listing of the current working directory (see **Notes**).

Pascal

```
procedure execvp1;

type
  %include /usr/include/ailtypes.inc
var
  merlin : integer;
  name : st80;
  args : pasargv;

%include /usr/include/aildefs.inc
```

```
begin
  name := 'examp';
  args[1] := 'examp';
  args[2] := '-x';
  args[3] := -F';
  args[4] := '-f';
  args[5] := '';
  merlin := p_execvp (name, args)
end;
```

FORTRAN

```
      SUBROUTINE EXECVP1
      INTEGER MERLIN
      CHARACTER*80 NAME, ARGS (80)
      NAME = 'examp '
      ARGS(1) = 'examp '
      ARGS(2) = '-x '
      ARGS(3) = '-F '
      ARGS(4) = '-f '
      ARGS(5) = ' '
      MERLIN = EXECVP (NAME, ARGS)
      END
```

Notes

The executable file 'examp' must be in the current directory before these examples will work.

EXIT, EEXIT
— terminate a process

Description

The **EXIT** system call is the normal means of terminating a process.

The **EEXIT** call exits the process without performing any of the clean-up operations performed by the **EXIT** routine.

Syntax

Pascal

```
p_exit (status);

p_eexit (status);
```

FORTRAN

```
EXIT (STATUS)

EEXIT (STATUS)
```

Parameters

status
is the termination status returned to the parent process.

- In Pascal, *status* is of type integer.
- In FORTRAN, *status* is of type INTEGER.

Return Values

There is no return value from a successful **EXIT** or **EEXIT** call.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **EXIT**, **FORK**, and **WAIT** system routines. Both create a child process, which issues the **EXIT** call. The parent process executes a **WAIT** call, and the parameter of that call ("green") receives the low-order eight bits of the value that the child passes to the **EXIT** routine. It is this value that is printed.

Pascal

```
procedure exit1;

type
  %include /usr/include/ailtypes.inc

var
  red, green, yellow, blue : integer;

%include /usr/include/aildefs.inc

begin
  green := p_fork;
  if green = 0 then
     blue := p_exit (red);
  yellow := p_wait (green);
  writeln ('status  ', green)
end;
```

FORTRAN

```
      SUBROUTINE EXIT1
      INTEGER EXIT, FORK, WAIT, BLUE, GREEN, RED, YELLOW
      GREEN = FORK ()
      IF (GREEN .EQ. 0) THEN
         BLUE = EXIT (RED)
      ENDIF
      YELLOW = WAIT (GREEN)
      PRINT *, 'STATUS ', GREEN
      END
```

Notes

In the AIX Operating System, **EEXIT** is named **__EXIT**.

FORK
— create a process

Description

The **FORK** system call creates a new process whose memory image is a copy of the memory image of the process that issued the **FORK** call.

Syntax

Pascal

```
p_fork;
```

FORTRAN

```
FORK ()
```

Parameters

This system call has no parameters.

Return Values

Upon successful completion, **FORK** returns the value 0 (zero) to the child process and the process ID of the child to the parent. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the next page call the **FORK** system routine to create a new process. The process ID of the child is returned to the parent process in the variable "blue", and the value 0 (zero) to the child process. Therefore both 0 and the process ID of the child are printed out.

Pascal

```
procedure fork1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;

%include /usr/include/aildefs.inc

begin
  blue := p_fork;
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE FORK1
      INTEGER FORK, BLUE
      BLUE = FORK ()
      PRINT *, BLUE
      END
```

NICE
— set a process priority

Description

The **NICE** system call assigns a new CPU priority to a process by adding a specified value to its current **NICE** value.

If this value results in a priority number outside the valid range, the **NICE** routine will reset the priority to the nearest limit.

Syntax

Pascal

```
p_nice (incr);
```

FORTRAN

```
NICE (INCR)
```

Parameters

incr

is a value that—when added to the priority number of the current process—determines the new priority number of the current process.

- In Pascal, *incr* is of type integer.
- In FORTRAN, *incr* is of type INTEGER.

Return Values

The new **NICE** value minus 20 is the value returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **NICE** system routine. The priority number of the current process is increased by 5, thereby lowering the priority. The *incr* parameter is specified with the variable "red". The return value printed is the new priority value minus 20.

Pascal

```
procedure nice1;

type
   %include /usr/include/ailtypes.inc
var
   red, blue : integer;

%include /usr/include/aildefs.inc

begin
   red := 5;
   blue := p_nice (red);
   writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE NICE1
      INTEGER NICE, RED, BLUE
      RED = 5
      BLUE = NICE (RED)
      PRINT *, BLUE
      END
```

PIPE
— create an interprocess channel

Description

The **PIPE** system call creates an interprocess communication mechanism—called a "pipe" or channel—that allows the passing of data between processes. After a pipe has been set up, two or more cooperating processes (created by subsequent **FORK** routines) can pass data to one another with **READ** and **WRITE** calls.

Syntax

Pascal

```
p_pipe (fildes);
```

FORTRAN

```
PIPE (FILDES)
```

Parameters

fildes
is an array of two file descriptors, both of which are returned by a **PIPE** call. The first element of the array holds the file descriptor for the read end of the pipe; the second element holds the file descriptor for the write end of the pipe.

- In Pascal, *fildes* is a variable of type piparray.
- In FORTRAN, *fildes* is an array(2) of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails (for example, if too many files are open).

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **PIPE** system routine. A pipe is created between two files whose descriptors are returned: the read end of the pipe is returned in the first element of the array "red" and the write end is returned in the second.

Pascal

```
procedure pipe1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;
  red : piparray;

%include /usr/include/aildefs.inc

begin
  blue := p_pipe (red);
  writeln (blue);
  writeln (red[1]);
  writeln (red[2])
end;
```

FORTRAN

```
      SUBROUTINE PIPE1
      INTEGER PIPE, BLUE, RED(2)
      BLUE = PIPE (RED)
      PRINT *, BLUE
      PRINT *, RED(1)
      PRINT *, RED(2)
      END
```

PLOCK
— lock or unlock a process, text, or data

Description

The **PLOCK** system call allows the calling process to lock or unlock its text segment (text lock), its data segment (data lock), or both (process lock) into memory. Locked segments are "pinned" in memory and are unaffected by paging.

Note: Only the super-user can issue this call.

Syntax

Pascal

```
p_plock (op);
```

FORTRAN

```
PLOCK (OP)
```

Parameters

op

is a constant or variable that specifies one of four options:

0 (UNLOCK) remove the locks.

1 (PROCLOCK) lock text and data segments into memory.

2 (TEXTLOCK) lock text segment into memory.

4 (DATALOCK) lock data segment into memory.

- In Pascal, *op* is of type integer.
- In FORTRAN, *op* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Example

The Pascal procedure and FORTRAN subroutine shown here call the **PLOCK** system routine. The value of the *op* parameter ("red") specifies that the routine lock the current text segment into memory.

Pascal

```
procedure plock1;

type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := 2;
  blue := p_plock (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE PLOCK1
      INTEGER PLOCK, RED, BLUE
      RED = 2
      BLUE = PLOCK (RED)
      PRINT *, BLUE
      END
```

WAIT
– delay the calling process

Description

The **WAIT** system call causes the calling process to delay until a signal is received or until one of its child processes terminates or stops in a trace mode. However, the routine does not delay the calling process if a child process that has *not* been waited for has already stopped or terminated before the call was issued.

If there are no children, the result is an error condition (see **Return Values**).

Syntax

Pascal

```
p_wait (stinfo);
```

FORTRAN

```
WAIT (STINFO)
```

Parameters

stinfo
is the termination status returned by one of the child processes to the parent process.

- In Pascal, the termination status is of type integer.
- In FORTRAN, the termination status is of type INTEGER.

Return Values

The process ID of a stopped or terminated child process is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **WAIT** system routine as well as two others that are commonly used in the context of a wait call: **FORK** and **EXECL**.

In both examples, the result is the creation of a new process that is a copy of the parent process. The **WAIT** call allows the inner loop of the child process to complete execution before the parent process proceeds further. Without the **WAIT** call, it is likely that the child process cannot complete the inner loop before the parent issues the **EXECL** call and prints the date. The **WAIT** call guarantees that the child process will complete the loop before the **EXECL** call is issued.

Pascal

```
procedure wait1;

type
  %include /usr/include/ailtypes.inc
var
  red, orange, yellow, green, blue, pink, purple : integer;

%include /usr/include/aildefs.inc
```

```
begin
  green := p_fork;
  if green = 0 then
     begin
       for orange := 1 to 40 do
         writeln ('child process');
       purple := p_exit (pink)
     end;
  blue := p_wait (red);
  writeln (blue);
  yellow := p_execl ('/bin/sh', 'sh', '-c', 'date', '')
end;
```

FORTRAN

```
      SUBROUTINE WAIT1
      INTEGER FORK, RED, ORANGE, YELLOW, GREEN, BLUE
      INTEGER PINK, PURPLE, WAIT, EXIT
      GREEN = FORK ()
      IF (GREEN .EQ. 0) THEN
         DO 10 ORANGE = 1,40
            PRINT *, 'CHILD PROCESS'
10    CONTINUE
      PURPLE = EXIT (PINK)
      ENDIF
      BLUE = WAIT (RED)
      PRINT *, BLUE
      YELLOW = EXECL ('/bin/sh ', 'sh ', '-c ', 'date ', ' ')
      END
```

Chapter 3. Process Identification

The following system calls can be used to get and set the different types of process identifiers, as well as get and set the limits of a process.

Call	Function
GETGRP	get the group access list.
GETUID, GTEUID, GETGID, GTEGID	get the real user ID, the real group ID, the effective user ID, and the effective group ID, respectively.
GTPGRP, GETPID, GTPPID	get the process group ID, the process ID, and the parent process ID, respectively.
SETGRP	set the group access list.
SETUID, SETGID	set user IDs (real and effective) and group IDs (real and effective), respectively.
STPGRP	set the process group ID.
ULIMIT	get and set process limits.
USRINF	get and set user information.

GETGRP
— get a group access list

Description

The **GETGRP** system call gets the group access list of the current process and stores it in an array specified in the call.

Syntax

Pascal

```
p_getgrp (ngrps, gidset);
```

Pascal external function definition

```
function p_getgrp (ngrps : integer;
                   gidset : getptr) : integer; external;
```

FORTRAN

```
GETGRP (NGRPS, GIDSET)
```

Parameters

ngrps
is the number of entries that can be stored in the array specified by the *gidset* parameter.

- In Pascal, *ngrps* is of type integer.
- In FORTRAN, *ngrps* is of type INTEGER.

gidset
points to an array in which the requested list items will be put. The maximum number of elements the array may hold is equal to the constant NGROUP defined in the Pascal and FORTRAN constants include files.

- In Pascal, *gidset* is a pointer of type getptr. (Getptr is a pointer to a user-defined integer array.)

- In FORTRAN, *gidset* is a user-defined array of type INTEGER.

Return Values

The value returned upon successful completion of the call is the number of elements stored in the group access list. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.

- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **GETGRP** system routine, which in these example returns a number that is equal to the number of elements in the array pointed to (Pascal) or specified (FORTRAN) by the variable "red". Note that, in Pascal, "red" is the user-defined array (of type getary) pointed to by getptr.

Pascal

```
procedure getgrp1;

type
  %include /usr/include/ailtypes.inc
  getary = array[1..20] of integer;
  getptr = @getary;
var
  blue, green : integer;
  red : getptr;

function p_getgrp (ngrps : integer;
                   gidset : getptr) : integer; external;

begin
  new (red);
  green := 20;
  blue := p_getgrp (green, red);
  writeln (blue)
end;
```

GETGRP

FORTRAN

```
SUBROUTINE GETGRP1
INTEGER GETGRP, BLUE, RED(20), GREEN
GREEN = 20
BLUE = GETGRP (GREEN, RED)
PRINT *, BLUE
END
```

Notes

In the AIX Operating System, **GETGRP** is named **getgroup**.

GETUID, GTEUID, GETGID, GTEGID
— get a user or group identifier

Description

The four **GET** system calls described in this section return the real or effective ID of a user or group.

- **GETUID** returns the ID of the real user of the calling process.
- **GTEUID** returns the effective user ID of the calling process.
- **GETGID** returns the real group ID of the calling process.
- **GTEGID** returns the effective group ID of the calling process.

Syntax

Pascal

```
p_getuid;

p_gteuid;

p_getgid;

p_gtegid;
```

FORTRAN

```
GETUID ()

GTEUID ()

GETGID ()

GTEGID ()
```

Parameters

These system calls have no parameters.

Return Values

The return value of each of the four calls is a particular ID (see description above).

- In Pascal, the return value is of type ushrt.
- In FORTRAN, the return value is of type INTEGER*2.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **GETGID** system routine, which returns the real group ID of the calling process in the variable "blue".

Pascal

```
procedure getgid1;

type
  %include /usr/include/ailtypes.inc
var
  blue : ushrt;

%include /usr/include/aildefs.inc

begin
  blue := p_getgid;
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE GETGID1
      INTEGER*2 GETGID, BLUE
      BLUE = GETGID ()
      PRINT *, BLUE
      END
```

Notes

In the AIX Operating System, **GTEUID** and **GTEGID** are named **geteuid** and **getegid**, respectively.

GTPGRP, GETPID, GTPPID
— get a process-group or process identifier

Description

The **GET** system calls described in this and the following section return the ID of a group, process, or user.

- **GTPGRP** returns the process group ID of the calling process.
- **GETPID** returns the process ID of the calling process and is often used to generate uniquely named temporary files.
- **GTPPID** returns the process ID of the parent process.

Syntax

Pascal

```
p_gtpgrp;

p_getpid;

p_gtppid;
```

FORTRAN

```
GTPGRP ()

GETPID ()

GTPPID ()
```

Parameters

These system calls have no parameters.

Return Values

The return value of each of the three calls is a particular ID (see description above).

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **GETPID** system routine, which returns the process ID in the variable "blue".

Pascal

```
procedure getpid1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;

%include /usr/include/aildefs.inc

begin
  blue := p_getpid;
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE GETPID1
      INTEGER GETPID, BLUE
      BLUE = GETPID ()
      PRINT *, BLUE
      END
```

Notes

In the AIX Operating System, **GTPGRP** and **GTPPID** are named **getpgrp** and **getppid**, respectively.

SETGRP
— set a group access list

Description

The **SETGRP** system call sets, or creates, the group access list of the current user process according to the values set in an array specified in the call.

Note: Only the super-user may issue this call.

Syntax

Pascal

```
p_setgrp (ngrps, gidset);
```

Pascal external function definition

```
function p_setgrp (ngrps : integer;
                   gidset : setptr) : integer; external;
```

FORTRAN

```
SETGRP (NGRPS, GIDSET)
```

Parameters

ngrps
is the number of entries in the array pointed to by *gidset*. This number may not exceed the constant NGROUP defined in the Pascal and FORTRAN constants include files.

- In Pascal, *ngrps* is of type integer.
- In FORTRAN, *ngrps* is of type INTEGER.

gidset
points to an array containing the values to be placed in the group access list. The maximum number of elements the array may hold is equal to the constant NGROUP defined in the Pascal and FORTRAN constants include files.

- In Pascal, *gidset* is a pointer of type setptr. (Setptr is a pointer to a user-defined integer array.)
- In FORTRAN, *gidset* is a user-defined array of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SETGRP** system routine, which in these examples sets the group access list of the current user process to that of the three named elements of the array pointed to (Pascal) or specified (FORTRAN) by the variable "red". Note that, in Pascal, "red" is the user-defined array (of type setary) pointed to by setptr.

Pascal

```
procedure setgrp1;

type
  %include /usr/include/ailtypes.inc
  setary = array[1..3] of integer;
  setptr = @setary;
var
  blue, green : integer;
  red : setptr;

function p_setgrp (ngrps : integer;
                   gidset : setptr) : integer; external;
```

```
begin
  new (red);
  red@[1] := 1;
  red@[2] := 2;
  red@[3] := 3;
  green := 3;
  blue := p_setgrp (green, red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE SETGRP1
      INTEGER SETGRP, BLUE, RED(3), GREEN
      RED(1) = 1
      RED(2) = 2
      RED(3) = 3
      GREEN = 3
      BLUE = SETGRP (GREEN, RED)
      PRINT *, BLUE
      END
```

Notes

In the AIX Operating System, **SETGRP** is named **setgroups**.

SETUID, SETGID
— set user or group identifiers

Description

The **SET** system calls described in this section set the user or group IDs to values specified in the call. Both the effective and the real IDs are set.

Syntax

Pascal

```
p_setuid (uid);

p_setgid (gid);
```

FORTRAN

```
SETUID (UID)

SETGID (GID)
```

Parameters

uid

is used with **SETUID**. It is the new value of the user ID to be set.

- In Pascal, *uid* is of type integer.
- In FORTRAN, *uid* is of type INTEGER.

gid

is used with **SETGID**. It is the new value of the new group ID to be set.

- In Pascal, *gid* is of type integer.
- In FORTRAN, *gid* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of a call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SETGID** system routine, which sets the real and effective group IDs. In these examples a value is obtained through a call to **GETGID** and then sent to **SETGID**.

Pascal

```
procedure setgid1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;
  red : ushrt;

%include /usr/include/aildefs.inc

begin
  red := p_getgid;
  blue := p_setgid (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE SETGID1
      INTEGER GETGID*2, RED, SETGID, BLUE
      RED = GETGID ()
      BLUE = SETGID (RED)
      PRINT *, BLUE
      END
```

STPGRP
— set a process group ID

Description

The **STPGRP** system call—when issued by a process—sets its process group ID to its process ID.

Syntax

Pascal

```
p_stpgrp;
```

FORTRAN

```
STPGRP ()
```

Parameters

This system call has no parameters.

Return Values

The new process group ID is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **STPGRP** system routine, which returns a new process group ID in the variable "blue".

Pascal

```
procedure stpgrp1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;

%include /usr/include/aildefs.inc

begin
  blue := p_stpgrp;
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE STPGRP1
      INTEGER STPGRP, BLUE
      BLUE = STPGRP ()
      PRINT *, BLUE
      END
```

Notes

In the AIX Operating System, **STPGRP** is named **setpgrp**.

ULIMIT
— get and set process limits

Description

The **ULIMIT** system call controls the limits of a process file.

Syntax

Pascal

```
p_ulimit (cmd, newlim);
```

FORTRAN

```
ULIMIT (CMD, NEWLIM)
```

Parameters

cmd
is a constant or a variable that can have one of the following values:

1 gets the process file-size limit.

2 sets the limit of the file size of the process to the value of *newlim* (see next parameter).

Note: Any process may decrease the limit, but only a process with an effective user ID of super-user may increase the limit.

3 retrieves the maximum possible break value (see **BRK** on page 2-2).

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

newlim
is used only with *cmd* option 2 to increment the limit.

- In Pascal, *newlim* is of type integer.
- In FORTRAN, *newlim* is of type INTEGER.

Return Values

A nonnegative value is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **ULIMIT** system routine, which in these examples returns the maximum possible break value (specified by the parameter value of 3) in the variable "blue".

Pascal

```
procedure ulimit1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;

%include /usr/include/aildefs.inc

begin
  blue := p_ulimit (3, 0);
  writeln (blue)
end;
```

FORTRAN

```
SUBROUTINE ULIMIT1
INTEGER ULIMIT, BLUE
BLUE = ULIMIT (3, 0)
PRINT *, BLUE
END
```

USRINF
– get and set user information

Description

The **USRINF** system call gets and sets information about the owner of the calling process.

Syntax

Pascal

```
p_usrinf (cmd, buf, count);
```

Pascal external function definition

```
function p_usrinf (cmd : integer; buf : usrptr;
                   count : integer) : integer; external;
```

FORTRAN

```
USRINF (CMD, BUF, COUNT)
```

Parameters

cmd

is a constant or a variable with two possible arguments (SETUINF or GETUINF) as defined in the Pascal and FORTRAN constants include files.

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

buf

is a pointer to a user buffer. The length of this buffer, in bytes, is usually equal to the constant INFSIZ(64).

- In Pascal, *buf* is a pointer of type usrptr. (Usrptr is a pointer to a user-defined packed array of type char.)
- In FORTRAN, *buf* is a user-defined variable of type CHARACTER.

count
is the number of bytes to be copied from or to the user buffer.

- In Pascal, *count* is of type integer.
- In FORTRAN, *count* is of type INTEGER.

Return Values

A nonnegative number indicating the number of bytes read is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **USRINF** system routine, which gets information about the owner of the current process. In these examples, the information is written to the array pointed to (Pascal) or specified by (FORTRAN) the variable "yellow". The number of bytes written to the array is returned in the variable "blue". Note that, in Pascal, "yellow" is the user-defined array (of type usrary) pointed to by usrptr.

Pascal

```
procedure usrinf1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
  usrary = packed array[1..INFSIZ] of char;
  usrptr = @usrary;
var
  red, blue : integer;
  yellow : usrptr;

function p_usrinf (cmd : integer; buf : usrptr;
                   count : integer) : integer; external;

begin
   new (yellow)
  blue := p_usrinf (GETINF, yellow, INFSIZ);
  writeln (blue);
  for red := 1 to blue do
    write (yellow@[red]);
    writeln
end;
```

FORTRAN

```
      SUBROUTINE USRINF1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER USRINF, BLUE
      CHARACTER*64 YELLOW
      BLUE = USRINF (GETINF, YELLOW, INFSIZ)
      PRINT *, BLUE
      PRINT *, YELLOW(1:BLUE)
      END
```

Chapter 4. Process Tracking

The following system calls can be used to monitor the operation of a process or a group of processes in the system.

Call	Function
ACCT	turns process accounting on or off.
PROFIL	generates an execution-time profile.
PTRACE	traces the execution of a process.
TIMES	gets process and child-process times.

ACCT
— turn process accounting on or off

Description

The **ACCT** call writes records in a specified "accounting file" whenever a process is terminated. Records of the terminated process are appended to the accounting file.

Note: Only users with an effective user ID of super-user may issue this call.

Syntax

Pascal

```
p_acct (path);
```

FORTRAN

```
ACCT (PATH)
```

Parameters

path
is the name of the file to which all accounting records are written. Passing the file name as an argument in the call activates the accounting function. Passing a null string turns the accounting function off.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **ACCT** routine. The accounting function is turned on and the records are appended to the files specified by *path*. The return value of the call is in "blue".

Pascal

```
procedure acct1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;
  red : st80;

%include /usr/include/aildefs.inc

begin
  red := '/tmp/acct';
  blue := p_acct (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE ACCT1
      INTEGER ACCT, BLUE
      CHARACTER*80 RED
      RED = '/tmp/acct '
      BLUE = ACCT (RED)
      PRINT *, BLUE
      END
```

PROFIL
— generate an execution-time profile

Description

The **PROFIL** system call generates a histogram of periodically sampled values of the program counter of the calling process.

Syntax

Pascal

```
p_profil (buf, bufsiz, offset, scale);
```

FORTRAN

```
PROFIL (BUF, BUFSIZ, OFFSET, SCALE)
```

Parameters

buf
is a pointer to the buffer in which the program counts are placed. Its length in bytes is specified by *bufsiz*.

- In Pascal, *buf* is of type shrtptr.
- In FORTRAN, *buf* is of type INTEGER*2.

bufsiz
specifies the size of the buffer in bytes. A value of 0 (zero) renders the routine ineffective. (See **Notes**.)

- In Pascal, *bufsiz* is of type usign.
- In FORTRAN, *bufsiz* is of type INTEGER.

offset

specifies the value to be subtracted from the program counter. (See **Notes.**)

- In Pascal, *offset* is of type usign.
- In FORTRAN, *offset* is of type INTEGER.

scale

specifies the value by which the quantity (program count – *offset*) is multiplied before the value in *buf* is incremented. (See **Notes.**)

- In Pascal, *scale* is of type usign.
- In FORTRAN, *scale* is of type INTEGER.

Return Values

There is no return value from a successful **PROFIL** call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **PROFIL** system routine. With the values assigned in the example, all instructions will be mapped to the area in memory pointed to by the variable "yellow".

Pascal

```
procedure profil1;

type
  %include /usr/include/ailtypes.inc
var
  yellow : shrtptr;
  blue, indigo, violet : usign;
  green : integer;

%include /usr/include/aildefs.inc
```

```
begin
   new (yellow);
  blue := 2;
  indigo := 0;
  violet := 1;
  green := p_profil (yellow, blue, indigo, violet)
end;
```

FORTRAN

```
      SUBROUTINE PROFIL1
      INTEGER PROFIL, YELLOW*2, GREEN, BLUE, INDIGO, VIOLET
      BLUE = 2
      INDIGO = 0
      VIOLET = 1
      GREEN = PROFIL (YELLOW, BLUE, INDIGO, VIOLET)
      END
```

Notes

Because Pascal and FORTRAN lack the facilities for handling unsigned four-byte integers, the programmer must make the appropriate conversion for parameter values that fall in the range

2 147 483 648 through 4 294 067 295

To use a parameter value in this range, subtract 4 294 067 296 from the parameter value before issuing the call (the result will always be negative).

PTRACE
— trace the execution of a child process

Description

The **PTRACE** routine enables a parent process to control the execution of a child process and to examine and change its memory image. The routine is used primarily for breakpoint debugging.

Syntax

Pascal

```
p_ptrace (requst, pid, addr, data, buff);
```

FORTRAN

```
FUNCTION PTRACE (REQUST, PID, ADDR, DATA, BUFF)
```

Parameters

requst
is a variable that specifies a trace operation (see *AIX Operating System Technical Reference*).

- In Pascal, *requst* is of type integer.
- In FORTRAN, *requst* is of type INTEGER.

pid
is a variable that contains the process ID of the traced process. This process must be an immediate child of the tracing process.

- In Pascal, *pid* is of type integer.
- In FORTRAN, *pid* is of type INTEGER.

addr

Depending on the value of *requst*, this parameter

- points to an area where data is returned; or
- indicates a register whose value is to be modified or returned; or
- points to a block of data (in the child process) to be read from or written to.

- In Pascal, *addr* is of type integer.
- In FORTRAN, *addr* is of type INTEGER.

data

when it is not ignored, usually holds data for requests that write to the memory image of the traced process.

- In Pascal, *data* is of type integer.
- In FORTRAN, *data* is of type INTEGER.

buff

is a pointer to a block of data (for any *requst* that requires a buffer).

- In Pascal, *buff* is of type integer.
- In FORTRAN, *buff* is of type INTEGER.

Return Values

For the values that are returned by **PTRACE**, see the descriptions of the arguments to the *requst* parameter. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **PTRACE** system routine. A child process is created by a **FORK** system call. The child process then calls **PTRACE**, requesting that it be traced by the parent (*requst* = 0). The parent process waits for a signal from the child and then calls **PTRACE**, which returns the value of register 2 used by the child process (*requst* = 11).

Pascal

```
procedure ptrace1;

type
  %include /usr/include/aildefs.inc
var
  blue, green, red, orange, yellow : integer;

%include /usr/include/ailtypes.inc

begin
  green := p_fork;
  if green = 0 then
     begin
        orange := p_alarm (1);
        orange := p_ptrace (0, 0, 0, 0, 0);
        for blue := 1 to 10 do
           for red := 1 to 100 do
              write ('z');
        writeln
     end

  else
     begin
        orange := p_wait (yellow);
        writeln ('return from wait   ', orange);
        orange := p_ptrace (11, green, 0, 2, 0);
        writeln ('register two contains   ', orange)
     end
end;
```

FORTRAN

```
      SUBROUTINE PTRACE1
      INTEGER RED, ORANGE, YELLOW, GREEN
      INTEGER BLUE, PTRACE, FORK, ALARM, WAIT
      GREEN = FORK ()

      IF (GREEN .EQ. 0) THEN
         ORANGE = ALARM (1)
         ORANGE = PTRACE (0, 0, 0, 0, 0)
         DO 10 BLUE = 1, 10
            DO 20 RED = 1, 100
               PRINT *, 'z'
20          CONTINUE
10       CONTINUE

      ELSE
         ORANGE = WAIT (YELLOW)
         PRINT *, 'RETURN FROM WAIT    ', ORANGE
         ORANGE = PTRACE (11, GREEN, 0, 2, 0)
         PRINT *, 'REGISTER TWO CONTAINS    ', ORANGE
      ENDIF
      END
```

TIMES
– get the process times

Description

The **TIMES** system call returns time-accounting information about the current process and about the terminated child processes of the current process.

Syntax

Pascal

```
p_times (buf);
```

FORTRAN

```
TIMES (BUF)
```

Parameters

buf
is a pointer to a data structure in which information about the current process times is placed.

- In Pascal, *buf* is of type tms.
- In FORTRAN, *buf* is an array(4) of type INTEGER.

Return Values

The elapsed time from a system-defined reference date to the current process time is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here issue a **TIMES** system call. A child process is created by a call to **FORK**. The return value is in the variable "green". The call to **TIMES** stores information in the buffer "colors". Both examples print the value in the tms__stime field, which is the CPU time used by the system on behalf of the calling process.

Pascal

```
procedure times1;

type
  %include /usr/include/ailtypes.inc
var
  colors : tms;
  red, green, blue : integer;

%include /usr/include/aildefs.inc

begin
  green := p_fork;
  if green = 0 then
    red := p_execl ('/bin/sh', 'sh', '-c' , 'date', '')
  else
    begin
      blue := 0 ;
      green := p_wait (blue);
      red := p_times (colors);
      writeln ('stime     ', colors.tms_stime)
    end
end;
```

FORTRAN

```
      SUBROUTINE TIMES1
      INTEGER TIMES, FORK, COLORS(4), RED, GREEN, BLUE
      GREEN = FORK ()
      IF (GREEN .EQ. 0) THEN
         RED = EXECL ('/bin/sh ', 'sh ', '-c ', 'date ', ' ')
      ELSE
         BLUE = 0
         GREEN = WAIT (BLUE)
         RED = TIMES (COLORS)
         PRINT *, 'stime     ', COLORS(2)
      ENDIF
      END
```

Chapter 5. Input and Output

The following system calls can be used to control data handling in the system. The operations performed include the creating, opening, and closing of files, and the movement of data into and out of them.

Call	Function
ACCESS	checks a file's accessibility.
CLOSE	closes a file.
CREAT	creates a new file or rewrites an existing one.
DUP	duplicates an open-file descriptor.
FCLEAR	clears space in a file.
FSYNC	writes changes in a file to permanent storage.
IOCTL	controls the input and output of a device.
LOCKF	locks a region of a file for exclusive access.
LSEEK	sets the read or write pointer.
OPEN	opens a file for reading or writing.
READ, READX	read data from a file or a device.
WRITE, WRITEX	write data to a file or device.

ACCESS
— check file accessibility

Description

The **ACCESS** system call checks a file's accessibility against a specified access mode.

Syntax

Pascal

```
p_access (path,amode);
```

FORTRAN

```
ACCESS (PATH, AMODE)
```

Parameters

path
is the name of the file to be checked.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

amode
is the access mode of the file specified by *path*. The parameter value is that of one of the parameter options or is constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files.

0 (F_OK) searches for a file

1 (X_OK) tests for execute permission

2 (W_OK) tests for write permission

4 (R_OK) tests for read permission

Specifying access mode 0 (zero) tests whether the directories leading to a file can be searched and whether the file exists.

- In Pascal, *amode* is a variable or constant of type integer.
- In FORTRAN, *amode* is a variable or constant of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **ACCESS** system routine. The accessibility of the file specified by *path* ("blue") is tested. The specified file is found and tested for execution, write, and read permissions as specified by the ORed value 7, defined in the variable "red".

Pascal

```
procedure access1;

type
  %include /usr/include/ailtypes.inc
var
  blue : st80;
  red, green : integer;

%include /usr/include/aildefs.inc
```

```
begin
  red = 7;
  blue := '/tmp/myfile';
  green := p_access (blue, red);
  writeln (green)
end;
```

FORTRAN

```
      SUBROUTINE ACCESS1
      CHARACTER*80
      INTEGER RED, GREEN, ACCESS
      RED = 7
      BLUE = '/tmp/myfile '
      GREEN = ACCESS (BLUE, RED)
      PRINT *, GREEN
      END
```

CLOSE
— close a file

Description

The **CLOSE** system call closes a specified file.

Syntax

Pascal

```
p_close (fildes);
```

FORTRAN

```
CLOSE (FILDES)
```

Parameters

fildes
is the file descriptor returned by a **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** system call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **OPEN** and **CLOSE** system routines. The **OPEN** call returns a file descriptor in the variable "red". This descriptor is used to close the same file.

Pascal

```
procedure close1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/tmp/anyfile', RDONLY, 0);
  blue := p_close (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE CLOSE1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER BLUE, RED, OPEN, CLOSE
      RED = OPEN ('/tmp/anyfile ', RDONLY, 0)
      BLUE = CLOSE (RED)
      PRINT *, BLUE
      END
```

CREAT
— create a new file

Description

The **CREAT** system call creates a new file or calls up an existing file in preparation for rewriting.

Syntax

Pascal

```
p_creat (path, mode);
```

FORTRAN

```
CREAT (PATH, MODE)
```

Parameters

path
is the name of the file being created or rewritten.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

mode
is the access mode of the file being created or rewritten. (For a list of *modes* see **CHMOD** on page 6-3.)

- In Pascal, *mode* is of type integer.
- In FORTRAN, *mode* is of type INTEGER.

Return Values

The return value is the file descriptor of the file created. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **CREAT** system routine. The variable "green" defines the *path* parameter. The Pascal and FORTRAN constants include files contain definitions of constants for the modes available in **CREAT**. These files are given owner read permissions as specified by the variable "red".

Pascal

```
procedure creat1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;
  green : st80;

%include /usr/include/aildefs.inc

begin
  red := IREAD;
  green := '/tmp/test.1';
  blue := p_creat (green, red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE CREAT1
      INCLUDE (/usr/include/ailfconsts.inc)
      CHARACTER*80 GREEN
      INTEGER BLUE, RED, CREAT
      RED = IREAD
      GREEN = '/tmp/test.1 '
      BLUE = CREAT (GREEN, RED)
      END
```

DUP
— duplicate an open-file descriptor

Description

The **DUP** system call returns a new file descriptor for a specified file. The descriptor to be duplicated must be an existing descriptor returned by an **OPEN**, **PIPE**, **FCNTL**, or **CREAT** system call. The new file descriptor is synonymous with the existing one.

Syntax

Pascal

```
p_dup (fildes);
```

FORTRAN

```
DUP (FILDES)
```

Parameters

fildes
is the file descriptor to be duplicated.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* of type INTEGER.

Return Values

The return value is the duplicate file-descriptor. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here make calls to the **DUP** system routine, which returns a file descriptor in the variable "blue". The Pascal and FORTRAN constants include files contain definitions of constants for the modes available in **OPEN**.

Pascal

```
procedure dup1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/usr/include/ailtypes.inc', RDONLY, 0);
  blue := p_dup (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE DUP1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER DUP, OPEN, RED, BLUE
      RED = OPEN ('/usr/include/ailtypes.inc ', RDONLY, 0)
      BLUE = DUP (RED)
      PRINT *, BLUE
      END
```

FCLEAR
— clear space in a file

Description

The **FCLEAR** system call clears space (makes a "hole") in a file by writing binary zeros to a specified number of bytes in the file. This "zeroing" process begins at the current position of the seek pointer of the file specified in the call.

Syntax

Pascal

```
p_fclear (fildes, nbytes);
```

FORTRAN

```
FCLEAR (FILDES, NBYTES)
```

Parameters

fildes
is the file descriptor of the file in which space is being cleared. This descriptor is obtained from a **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** system call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

nbytes
is a constant or a variable specifying the number of bytes to be zeroed. If this number falls within a certain range, the programmer will have to use a conversion formula to obtain the proper value for *nbytes* (see **Notes**).

- In Pascal *nbytes* is of type usign.
- In FORTRAN *nbytes* is of type INTEGER.

Return Values

The return value is *nbytes*. If this value falls within a certain range, the programmer will have to use a conversion formula to obtain the actual number (see **Notes**).

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **FCLEAR** system routine, which overwrites the specified open file /tmp/junk with 200 null characters.

Pascal

```
procedure fclear1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/tmp/junk', WRONLY, 0);
  blue := p_fclear (red, 200);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE FCLEAR1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER RED, BLUE, OPEN, FCLEAR
      RED = OPEN ('/tmp/junk ', WRONLY, 0)
      BLUE = FCLEAR (RED, 200)
      PRINT *, BLUE
      END
```

Notes

Because Pascal and FORTRAN lack the facilities for handling unsigned four-byte integers, the programmer must make the following conversion of parameter values of type usign that fall in the range

2 147 483 648 through 4 294 067 295

To use a parameter value in this range, *subtract* 4 294 067 296 from the parameter value (the result will always be negative) before issuing the call.

Conversely, if the return value is a negative number, *add* 4 294 067 296 to that number to obtain the correct value.

FSYNC
— write to permanent storage

Description

The **FSYNC** system call writes all modified data in a specified open file to permanent storage.

Syntax

Pascal

```
p_fsync (fildes);
```

FORTRAN

```
FSYNC (FILDES)
```

Parameters

fildes
the open-file descriptor obtained by a **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** routine.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **FSYNC** system routine, which writes changes in a specified file to permanent storage that is mapped copy-on-write by means of the **SHMAT** call (see page 10-2).

Pascal

```
procedure fsync1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, blue, yellow : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/tmp/junk', WRONLY, 0);
  yellow := p_shmat (red, 0, SHMCPY);
  blue := p_fsync (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE FSYNC1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER FSYNC, OPEN, SHMAT, RED, BLUE, YELLOW
      RED = OPEN ('/tmp/junk ', WRONLY, 0)
      YELLOW = SHMAT (RED, 0, SHMCPY)
      BLUE = FSYNC (RED)
      PRINT *, BLUE
      END
```

IOCTL
— control the input and output of a device

Description

The **IOCTL** system call performs a variety of functions on both block and character special files (devices). (For information about available devices see *AIX Operating System Commands Reference* and *AIX Operating System Technical Reference.*)

Syntax

Pascal

```
p_ioctl (fildes, requst, arg);
```

FORTRAN

```
IOCTL (FILDES, REQUST, ARG)
```

Parameters

fildes
is the file descriptor of the opened device.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

requst
is either of two operations to be performed on the device specified by *fildes*. Both are defined in the Pascal and FORTRAN constants include files. They are as follows:

IOCTYP returns the device type associated with *fildes*. The device types are defined in the constant include files.

IOCINF stores device information specified by *fildes* in the buffer specified by *arg*.

- In Pascal, *requst* is of type integer.
- In FORTRAN, *requst* is of type INTEGER.

arg

is a data structure used to pass and receive values from the **IOCTL** routine.

- In Pascal, *arg* is of type devptr.

  Note: The Pascal type-definition file /usr/include/ailtypes.inc may have to be edited, and the data structure pointed to by devptr changed, to make that structure acceptable to the device specified in the call.

- In FORTRAN, *arg* is a variable or array of type INTEGER.

  Note: In FORTRAN, *arg* must be defined in the program to make it acceptable to the device specified in the call.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **IOCTL** system routine, which returns information about device /dev/lp in the Pascal record "green" or FORTRAN array "GREEN".

Pascal

```
procedure ioctl1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, yellow, blue : integer;
  green : devptr;

%include /usr/include/aildefs.inc

begin
  new (green);
  red := p_open ('/dev/lp', RDWR, 0);
  yellow := IOCINF;
  blue := p_ioctl (red, yellow, green);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE IOCTL1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER IOCTL, OPEN, RED, GREEN, BLUE, YELLOW
      RED = OPEN ('/dev/lp ', RDWR, 0)
      YELLOW = IOCNF
      BLUE = IOCTL (RED, YELLOW, GREEN)
      PRINT *, BLUE
      END
```

LOCKF
— lock or unlock a region of a file

Description

The **LOCKF** system call locks and unlocks regions of an open file. It is used to synchronize simultaneous access to a specified open file by multiple processes. Only one process at a time can maintain a "lock" on a region of a file. The **LOCKF** system call can invoke either of two kinds of lock: (1) enforced or (2) advisory.

1. When a process holds an *enforced* lock on a region of a file:

   a. no other process can access that region with read or write system calls; and

   b. **CREAT** and **OPEN** are prevented from truncating the file.

2. When a process holds an *advisory* lock on a region of a file:

   a. no other process can lock that region or an overlapping region with the **LOCKF** call; and

   b. the **CREAT**, **OPEN**, **READ**, and **WRITE** call are not affected, which means that a process itself must issue a **LOCKF** call in order to make advisory locks effective.

Note: To select enforced locking, the ENFMT access mode of the specified file must be set. Otherwise, locking is advisory. Thus a given file may have enforced locks or advisory locks but not both.

Warning: Buffered I/O does not work properly with file locking.

Syntax

Pascal

```
p_lockf (fildes, requst, size);
```

FORTRAN

```
LOCKF (FILDES, REQUST, SIZE)
```

Parameters

fildes
is the open-file descriptor obtained from a **CREAT**, **FCNTL**, **OPEN**, or **PIPE** system call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

requst
can be a constant or a variable and is defined by one of the following:

0 (F_ULOCK) unlocks a previously locked region in the file.

1 (F_LOCK) locks the region for exclusive use.

2 (F_TLOCK) tests to see if another process has locked the specified region and, if not, locks the region.

3 (F_TEST) tests to see if another process has already locked a region.

- In Pascal, *requst* is of type integer.
- In FORTRAN, *requst* is of type INTEGER.

size
can be a constant or a variable and it defines the number of bytes being locked or unlocked. Unallocated "holes" in the file can also be locked (see **FCLEAR** on page 5-12).

- In Pascal *size* is of type integer.
- In FORTRAN, *size* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **LOCKF** system routine, which locks an open file "forward" 1000 bytes.

Pascal

```
procedure lockf1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/usr/include/ailtypes.inc', RDONLY, 0);
  blue := p_lockf (red, 1, 1000);
  writeln (blue)
end;
```

FORTRAN

```
SUBROUTINE LOCKF1
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER LOCKF, OPEN, BLUE, RED
RED = OPEN ('/usr/include/ailtypes.inc ', RDONLY, 0)
BLUE = LOCKF (RED, 1, 1000)
PRINT *, BLUE
END
```

LSEEK
— set a read or write pointer

Description

The **LSEEK** system call sets a read or write pointer in a specified file that has been opened for reading or writing.

Syntax

Pascal

```
p_lseek (fildes, offset, whence);
```

FORTRAN

```
LSEEK (FILDES, OFFSET, WHENCE)
```

Parameters

fildes
is the descriptor of the file to be read from or written to. The file descriptor is the return value obtained from a **CREAT**, **DUP**, **FCNTL**, or **OPEN** call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

offset
is a value (number of bytes) used in combination with the *whence* parameter to position the pointer in the file.

- In Pascal, *offset* is of type integer.
- In FORTRAN, *offset* is of type INTEGER.

whence

specifies how the *offset* value will be used to position the file pointer of *fildes*.

0 the pointer will be set to the value of *offset*.

1 the pointer will be set to the value of the current location plus the *offset* value.

2 the pointer will be set to the value of the *offset* number of bytes plus the size of the file.

- In Pascal, *whence* is of type integer.
- In FORTRAN, *whence* is of type INTEGER.

Return Values

The return value is the new location of the file pointer as measured in bytes from the beginning of the file. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the next page call the **LSEEK** system routine, which moves the file pointer to the 200-byte mark of the open file specified in the call. The return value in "yellow" should in this case equal the offset of 200.

Pascal

```
procedure lseek1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, yellow, blue, green : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/usr/include/ailtypes.inc', RDONLY, 0);
  blue := 0;
  green := 200;
  yellow := p_lseek (red, green, blue);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE LSEEK1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER LSEEK, OPEN, RED, YELLOW, GREEN, BLUE
      RED = OPEN ('/usr/include/ailtypes.inc ', RDONLY, 0)
      BLUE = 0
      GREEN = 200
      YELLOW = LSEEK (RED, GREEN, BLUE)
      PRINT *, YELLOW
      END
```

OPEN
— open a file for reading or writing

Description

The **OPEN** system call opens a specified file for reading or writing or both, depending on the access mode specified in the call.

Syntax

Pascal

```
p_open (path, oflag, mode);
```

FORTRAN

```
OPEN (PATH, OFLAG, MODE)
```

Parameters

path
is the name of the file to be opened.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

oflag
specifies one or a combination of the options listed below. The parameter value is that of one of the following options or is constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files (see Appendixes).

Note: The RDONLY, WRONLY, and RDWR values *cannot* be logically ORed together.

RDONLY	opens the file for reading.
WRONLY	opens the file for writing.
RDWR	opens the file for both reading and writing.
NDELAY	open without delay. This flag may affect subsequent reads and writes.
APPEND	sets the file pointer to the end of the file prior to each write.
CREATE	has no effect if the file specified by *path* exists. However, if the specified file does not exist, the file owner's ID and the files's group ID are set to the effective user ID of the process; and the access mode is set to *mode*.
TRUNC	truncates the file length to zero.
EXCL	when this option and **CREATE** are set, **OPEN** will fail if the file exists.

- In Pascal, *oflag* is of type integer.
- In FORTRAN, *oflag* is of type INTEGER.

mode
is used with the **CREATE** value of *oflag*.

Note: For more information on the *mode* parameter, see **CHMOD** on page 6-3 and **STAT** on page 6-25.

Return Values

The return value is the file descriptor of the opened file. This file descriptor will be needed for subsequent input-output operations. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **OPEN** system routine, which returns a file descriptor in the variable "red". If the call is successful, the number printed out is a valid file descriptor.

```
procedure open1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red : integer;

%include /usr/include/aildefs.inc

begin
  red := p_open ('/usr/include/ailtypes.inc', RDONLY, 0);
  writeln(red)
end;
```

FORTRAN

```
      SUBROUTINE OPEN1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER OPEN, RED
      CHARACTER*80 BLUE
      BLUE = '/usr/include/ailtypes.inc '
      RED = OPEN (BLUE, RDONLY, 0)
      PRINT *, RED
      END
```

READ, READX
— read from a file

Description

The **READ** system call reads a specified number of bytes from a file into a buffer.

The **READX** system call invokes the same function as **READ**, but it provides the alternative of communication with character device drivers that require more information or return more status information than **READ** can handle.

Syntax

Pascal

```
p_read (fildes, buf, nbytes);

p_readx (fildes, buf, nbytes, ext);
```

Pascal external function definitions

```
function p_read (fildes : integer; buf : readptr;
                 nbytes : integer) : integer; external;

function p_readx (fildes : integer; buf : readptr;
                  nbytes, ext : integer) : integer; external;
```

FORTRAN

```
READ (FILDES, BUF, NBYTES)

READX (FILDES, BUF, NBYTES, EXT)
```

Parameters

fildes
is the open-file descriptor returned from a successful **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

buf
is a pointer to a buffer. The bytes read from the file specified by *fildes* are placed in this buffer when a **READ** or **READX** system call is executed.

- In Pascal, *buf* is a pointer of type readptr. (Readptr is a user-defined packed array of type character.)
- In FORTRAN, *buf* is a user-defined array of type CHARACTER.

nbytes
is the number of bytes to be read from the file specified by *fildes*.

- In Pascal, *nbytes* is of type integer.
- In FORTRAN, *nbytes* is of type integer.

ext
is a parameter of the **READX** call only. It provides a value or a pointer to a communication area for specific devices.

- In Pascal, *ext* is of type integer.
- In FORTRAN, *ext* is of type INTEGER.

In Pascal and FORTRAN, *ext* is device-dependent (see *AIX Operating System Technical Reference*).

Return Values

The return value is the actual number of bytes read from the file. If the return value is 0 (zero), the end file has been reached. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **READ** system routine, which reads a specified number of bytes from a file that has been opened for reading. In these examples, 100 bytes are read from the file /usr/include/ailtypes.inc into the buffer pointed to by the Pascal variable "yellow" and by the FORTRAN string "YELLOW".

Pascal

```
procedure read1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
  readary = packed array[1..100] of char;
  readptr = @readary;
var
  yellow : readptr;
  red, blue, orange : integer;

function p_read (fildes : integer; buf : readptr;
                 nbytes : integer) : integer; external;

begin
  new (yellow);
  blue := p_open ('/usr/include/ailtypes.inc', RDONLY, 0);
  red := 100;
  orange := p_read (blue, yellow, red);
  writeln (orange)
end;
```

FORTRAN

```
      SUBROUTINE READ1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER OPEN, READ, RED, ORANGE, BLUE
      CHARACTER*100 YELLOW
      BLUE = OPEN ('/usr/include/ailtypes.inc ', RDONLY, 0)
      RED = 100
      ORANGE = READ (BLUE, YELLOW, RED)
      PRINT *, ORANGE
      END
```

WRITE, WRITEX
— write to a file

Description

The **WRITE** system call writes a specified number of bytes from a specified area to a specified file.

The **WRITEX** system call invokes additional communications facilities.

Syntax

Pascal

```
p_write (fildes, buffer, nbytes);

p_writex (fildes, buffer, nbytes, ext);
```

Pascal external function definitions

```
function p_write (fildes : integer; buffer : writptr;
                  nbytes : integer) : integer; external;

function p_writex (fildes : integer; buffer : writptr;
                nbytes, ext : integer) : integer; external;
```

FORTRAN

```
WRITE (FILDES, BUFFER, NBYTES)

WRITEX (FILDES, BUFFER, NBYTES, EXT)
```

Parameters

fildes
is the file descriptor of the file to be written to. The value of the descriptor is returned from a successful **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

buffer
is a pointer to a buffer of *nbytes* contiguous bytes that are written to the output file. The number of characters actually written is returned. It should be regarded as an error if the return value differs from the number requested.

- In Pascal, *buffer* is a pointer of type writptr. (Writptr is a pointer to a user-defined packed array of type char.)
- In FORTRAN, *buffer* is a user-defined array of type CHARACTER.

nbytes
is the number of bytes to be written to the specified file.

- In Pascal, *nbytes* is of type integer.
- In FORTRAN, *nbytes* is of type INTEGER.

ext
is a parameter of the **WRITEX** call. It provides a value or a pointer to a communications area for specific devices.

- In Pascal, *ext* is of type integer.
- In FORTRAN, *ext* is of type INTEGER.

In Pascal and FORTRAN, *ext* is device-dependent (see *AIX Operating System Technical Reference*).

Return Values

The return value is the number of bytes written to the specified file. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **WRITE** system routine, which writes a specified number of bytes to a file that has been opened for writing. In these examples, 35 bytes are written to the file /tmp/junk from the Pascal packed array "yellow" and from the FORTRAN array "YELLOW".

Pascal

```
procedure write1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
  writary = packed array[1..35] of char;
  writptr = @writary;
var
  yellow : writptr;
  red, orange, blue : integer;

function p_write (fildes : integer; buf : writptr;
                  nbytes : integer) : integer; external;

begin
  new(yellow);
  yellow@ := 'test file for the WRITE system call';
  blue := p_open ('/tmp/junk', WRONLY, 0);
  red := 35;
  orange := p_write (blue, yellow, red);
  writeln (orange)
end;
```

FORTRAN

```
      SUBROUTINE WRITE1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER WRITE, OPEN, RED, ORANGE, BLUE
      CHARACTER*35 YELLOW
      BLUE = OPEN ('/tmp/junk ', WRONLY, 0)
      YELLOW = 'test file for the WRITE system call'
      RED = 35
      ORANGE = WRITE (BLUE, YELLOW, RED)
      PRINT *, ORANGE
      END
```

Chapter 6. File Maintenance

The following system calls can be used to change the access permissions of files, create directories, mount file systems, and perform a variety of other maintenance functions. Many of these calls are the basis for the system commands that have similar names. However, they can also be used to write new commands and utilities that can be included in application programs or used in program development.

Call	Function
CHMOD	changes file-access permissions.
CHOWN	changes the user and group ownership of a file.
CHROOT	changes the root directory.
FCNTL	controls an open-file descriptor.
FTRUNC	shortens a file.
LINK	link to a file.
MKNOD	creates a directory or special file.
MOUNT, UMOUNT	mounts or unmounts a file system.
STAT, FSTAT	get file-status information.
SYNC	updates a file system.
UMASK	gets and sets the file-creation-mode mask.

UNLINK	deletes a directory entry.
USTAT	gets file-system information.
UTIME	sets access and modification times of a file.

CHMOD
—change file-access permissions

Description

The **CHMOD** system call changes the access permissions, or *access mode*, of a specified file.

Note: Only the owner of a file and the super-user can change the access mode of that file.

Syntax

Pascal

```
p_chmod (path, mode);
```

FORTRAN

```
CHMOD (PATH, MODE)
```

Parameters

path
is the name of the file whose access mode is being changed.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

mode
is the new access mode for the file specified by *path*. The parameter value is that of one of the parameter options (see next page) or is constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files.

Constant	Access Attribute
ISUID	set user ID on execution
ISGID	set group ID on execution
ISVTX	save text image after execution
ENFMT	enables enforcement mode record locking
IRUSR	read by owner
IWUSR	write by owner
IXUSR	execute file (or search directory) by owner
IRGRP	read by group
IWGRP	write by group
IXGRP	execute by group
IROTH	read by others
IWOTH	write by others
IXOTH	execute by others

- In Pascal, *mode* is of type integer.
- In FORTRAN, *mode* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **CHMOD** system routine, which in these examples changes the access mode of "anyfile" to "read by others," specified by the attribute IROTH of the *mode* parameter ("red"). The file affected is assumed to be a valid file owned by the issuer of the call.

Pascal

```
procedure chmod1;

const
  %include/usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;
  green : st80;

%include /usr/include/aildefs.inc

begin
  red := IROTH;
  green := 'anyfile';
  blue := p_chmod ('anyfile', red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE CHMOD1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER CHMOD, RED, BLUE
      CHARACTER*80 GREEN
      RED = IROTH
      GREEN = 'anyfile '
      BLUE = CHMOD (GREEN, RED)
      PRINT *, BLUE
      END
```

CHOWN
— change ownership of a file

Description

The **CHOWN** system call changes the ownership of a specified file by changing the user and group IDs.

Note: Only the owner of a file and the super-user can change the ownership of that file.

Syntax

Pascal

```
p_chown (path, owner, group);
```

FORTRAN

```
CHOWN (PATH, OWNER, GROUP)
```

Parameters

path
is the name of the file whose owner and group IDs are being changed.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

owner
is the user ID of the new owner of the file specified by *path*.

- In Pascal, *owner* is of type integer.
- In FORTRAN, *owner* is of type INTEGER.

group
is the group ID of the new owner of the file specified by *path*.

- In Pascal, *group* is of type integer.
- In FORTRAN, *group* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **CHOWN** system routine, which in these examples assigns the ownership of "myfile" to the owner of root. The file affected is assumed to be a valid file owned by the issuer of the call.

Pascal

```
procedure chown1;

type
  %include /usr/include/ailtypes.inc
var
  blue, green, red : integer;
  yellow : st80;

%include /usr/include/aildefs.inc

begin
  red := 0;
  green := 0;
  yellow := 'myfile';
  blue := p_chown ('myfile', red, green);
  writeln (blue)
end;
```

FORTRAN

```
SUBROUTINE CHOWN1
INTEGER CHOWN, BLUE, GREEN, RED
CHARACTER*80 YELLOW
RED = 0
GREEN = 0
YELLOW = 'myfile '
BLUE = CHOWN (YELLOW, RED, GREEN)
PRINT *, BLUE
END
```

CHROOT
— change the root directory

Description

The **CHROOT** system call changes a specified directory to the effective root directory (the starting point when searching for pathnames that begin with "/").

Note: Only the super-user may issue this call.

Syntax

Pascal

```
p_chroot (path);
```

FORTRAN

```
CHROOT (PATH)
```

Parameters

path
is the name of the directory that will be used as the home directory for file names beginning with "/".

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **CHROOT** system routine, which in these examples makes /usr/include the effective root directory for the life of the calling process.

Pascal

```
procedure chroot1;

type
  %include /usr/include/ailtypes.inc
var
  blue : st80;
  red : integer;

%include /usr/include/aildefs.inc

begin
  blue := '/usr/include';
  red := p_chroot (blue);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE CHROOT1
      INTEGER CHROOT, BLUE, RED
      CHARACTER*80 BLUE
      BLUE = '/usr/include '
      RED = CHROOT (BLUE)
      PRINT *, RED
      END
```

FCNTL
— control an open-file descriptor

Description

The **FCNTL** system call performs various control operations on an open-file descriptor.

Syntax

Pascal

```
p_fcntl (fildes, cmd, arg);
```

FORTRAN

```
FCNTL (FILDES, CMDS, ARG)
```

Parameters

fildes
is the open-file descriptor obtained from a **CREAT**, **DUP**, **OPEN**, or **PIPE** system call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

cmd
is a variable or constant specifying the operation to be performed. The five command options for this routine are:

0 (F__DUPFD) returns a new file descriptor.

1 (F__GETFD) returns the value of the close-on-exec flag associated with the file descriptor *fildes*.

2 (F__SETFD) sets the close-on-exec flag associated with *fildes* to the value of the low-order bit of *arg*.

3 (F_GETFL) gets the file status flags of the file descriptor *fildes*.

4 (F_SETFL) sets the file status flags to the value of *arg*.

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

arg

is a constant or a variable whose value is 1 or 0 depending on whether or not the *cmd* parameter has been set to option 2.

- In Pascal, *arg* is of type integer.
- In FORTRAN, *arg* is of type INTEGER.

Return Values

The value returned varies according to the command option specified in the call:

0 (F_DUPFD) a new file descriptor

1 (F_GETFD) the value of the flag (only the low-order bit is defined)

2 (F_SETFD) a value other than -1

3 (F_GETFL) the value of file flags

4 (F_SETFL) a value other than -1

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **FCNTL** system routine, which in these examples opens the file /usr/include/ailtypes.inc for reading and writing. The file descriptor returned by the **OPEN** call is used for the *fildes* parameter ("blue") in **FCNTL**; the *cmd* parameter ("red") instructs the system to return the file-status flags of the file descriptor. This is the value printed out.

Pascal

```
procedure fcntl1;

type
  %include /usr/include/ailtypes.inc
var
  blue, green, red, yellow : integer;

%include /usr/include/aildefs.inc

begin
  red := 3;
  green := 0;
  blue := p_open ('/usr/include/ailtypes.inc', 2, 0);
  yellow := p_fcntl (blue, red, green);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE FCNTL1
      INTEGER FCNTL, OPEN, BLUE, GREEN, RED, YELLOW
      RED = 3
      GREEN = 0
      BLUE = OPEN ('/usr/include/ailtypes.inc ', 2, 0)
      YELLOW = FCNTL (BLUE, RED, GREEN)
      PRINT *, YELLOW
      END
```

FTRUNC
— truncate a file

Description

The **FTRUNC** system call counts a specified number of bytes from the beginning of a specified file and then deletes all the remaining bytes.

Syntax

Pascal

```
p_ftrunc (fildes, len);
```

FORTRAN

```
FTRUNC (FILDES, LEN)
```

Parameters

fildes
is an open-file descriptor.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

len
is the number of bytes to be left in the truncated file, counting from the first byte. (See **Notes.**)

- In Pascal, *len* is of type usign.
- In FORTRAN, *len* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **FTRUNC** system routine, which in these examples truncates the file /tmp/xxx (assuming that it exists) to a length of 100 bytes as specified by the *len* parameter ("blue").

Pascal

```
procedure ftrunc1;

type
  %include /usr/include/ailtypes.inc
var
  blue, red, yellow : integer;
  orange : st80;

%include /usr/include/aildefs.inc

begin
  orange := '/tmp/xxx';
  blue := 100;
  red := p_open (orange, 2, 0);
  yellow := p_ftrunc (red, blue);
  writeln (yellow)
end;
```

FORTRAN

```
SUBROUTINE FTRUNC1
INTEGER FTRUNC, OPEN, BLUE, RED, YELLOW
CHARACTER*80 ORANGE
ORANGE = '/tmp/xxx '
BLUE = 100
RED = OPEN (orange, 2, 0)
YELLOW = FTRUNC (RED, BLUE)
PRINT *, YELLOW
END
```

Notes

1. Because Pascal and FORTRAN lack the facilities for handling unsigned four-byte integers, the programmer must convert parameter values of type usign that fall in the range

   2 147 483 648 through 4 294 067 295

   To use a parameter value in this range, subtract 4 294 067 296 from that value before issuing the call (the result will always be negative).

2. In the AIX Operating System, this system call has the name **ftruncate**.

LINK
– link to a file

Description

The **LINK** system call creates a link to a specified file.

Syntax

Pascal

```
p_link (path1, path2);
```

FORTRAN

```
LINK (PATH1, PATH2)
```

Parameters

path1
is the name of the file to be linked.

- In Pascal, *path1* is a string variable or constant of type st80.
- In FORTRAN, *path1* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

path2
is the name of the new directory entry to be created.

- In Pascal, *path2* is a string variable or constant of type st80.
- In FORTRAN, *path2* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **LINK** system routine, which in these examples creates a second link (/tmp/new) for the file /tmp/xxx. This will *not* be a copy of the file /tmp/xxx but an additional link to the existing file.

Pascal

```
procedure link1;

type
  %include /usr/include/ailtypes.inc
var
  yellow : integer;
  red, blue : st80;

%include /usr/include/aildefs.inc

begin
  red := '/tmp/xxx';
  blue := '/tmp/new';
  yellow := p_link (red, blue);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE LINK1
      INTEGER LINK, YELLOW
      CHARACTER*80 BLUE, RED
      RED = '/tmp/xxx '
      BLUE = '/tmp/new '
      YELLOW = LINK (RED, BLUE)
      PRINT *, YELLOW
      END
```

MKNOD
— create a directory or special file

Description

The **MKNOD** system call creates a new regular file, special file, or directory; specifies an access mode that includes directory special-file bits; and initializes the first pointer of the i-node.

Note: Only the super-user may issue this call.

Syntax

Pascal

```
p_mknod (path, mode, dev);
```

FORTRAN

```
MKNOD (PATH, MODE, DEV)
```

Parameters

path
is the name of the new file or directory.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

mode
is the access mode of the new file and includes special bits and directory bits. (For the values and meaning of the various access attribute bits, see **CHMOD** on page 6-3.) The protection bits of the mode are modified by the process mode mask (see **UMASK** on page 6-30)

- In Pascal, *mode* is of type integer.
- In FORTRAN, *mode* is of type INTEGER.

dev

initializes the first block pointer of the i-node. For ordinary files and directories, *dev* is usually zero. In the case of a special file, *dev* specifies the file to be created. (For information on special-file bits, see *AIX Operating System Technical Reference*.)

- In Pascal, *dev* is of type integer.
- In FORTRAN, *dev* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER

Examples

The Pascal procedure and FORTRAN subroutine shown here call **MKNOD** system routine, which in these examples creates a file (/tmp/junk). The value of *mode* ("blue") specifies a text file with read and write privileges for the owner of the file.

Pascal

```
procedure mknod1;

type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;
  yellow : st80;

%include /usr/include/aildefs.inc
```

```
begin
  yellow := '/tmp/junk';
  blue := 33152;
  red := p_mknod (yellow, blue, 0);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE MKNOD1
      INTEGER MKNOD, BLUE, RED
      CHARACTER*80 YELLOW
      YELLOW = '/tmp/junk '
      BLUE = 33152
      RED = MKNOD (YELLOW, BLUE, 0)
      PRINT *, RED
      END
```

MOUNT, UMOUNT
— mount or unmount a file system

Description

The **MOUNT** system call mounts a removable file system on a block-structured special file, names a new root file for that file system, and specifies whether the system is write enabled or write protected.

The **UMOUNT** system call unmounts a removable file system: the associated root file is replaced by the default version, any pending I/O for the unmounted system is completed, and the system itself is marked clean.

Syntax

Pascal

```
p_mount (dev, dir, rwflag);

p_umount (dev);
```

FORTRAN

```
MOUNT (DEV, DIR, RWFLG)

UMOUNT (DEV)
```

Parameters

dev

specifies the device on which the file system is to be mounted or from which it is to be unmounted.

- In Pascal, *dev* is a string variable or constant of type st80.
- In FORTRAN, *dev* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

dir
is used only with **MOUNT**. It is the name of the directory of the file system that is to be mounted. The file specified by *dir* must exist and it must be a directory unless the root file of the mounted file system is not a directory.

- In Pascal, *dir* is a string variable or constant of type st80.
- In FORTRAN, *dir* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

rwflag
is used only with **MOUNT**. The least significant bit specifies whether the file system is write enabled or not.

- In Pascal, *rwflag* is of type integer.
- In FORTRAN, *rwflag* is of type INTEGER.

Return Values

MOUNT returns the value 0 (zero) upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

UMOUNT returns the value 0 (zero) upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the next page call the **UMOUNT** system routine. In these examples, the call instructs the routine to unmount a device.

Pascal

```
procedure umount1;

type
  %include /usr/include/ailtypes.inc
var
  yellow : integer;
  blue : st80;

%include /usr/include/aildefs.inc

begin
  blue := '/dev/hd9';
  yellow := p_umount (blue);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE UMOUNT1
      INTEGER UMOUNT, YELLOW
      CHARACTER*80 BLUE
      BLUE = '/dev/hd9 '
      YELLOW = UMOUNT (BLUE)
      PRINT *, YELLOW
      END
```

STAT, FSTAT
— return the status of a file

Description

The **STAT** system call obtains status information about a specified file.

The **FSTAT** system routine obtains status information about an open file identified by the file descriptor from a successful **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** call.

Syntax

Pascal

```
p_stat (path, buf);

p_fstat (fildes, buf);
```

FORTRAN

```
STAT (PATH, BUF)

FSTAT (FILDES, BUF)
```

Parameters

path
is used only in the **STAT** system call. It specifies the file whose status is to be checked.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

fildes
is used only in the **FSTAT** system call. It is an open-file descriptor obtained from a successful **CREAT**, **DUP**, **FCNTL**, **OPEN**, or **PIPE** call.

- In Pascal, *fildes* is of type integer.
- In FORTRAN, *fildes* is of type INTEGER.

buf
is required for both system calls. It points to a buffer where status information about the specified file is stored.

- In Pascal, *buf* is of type statptr.
- In FORTRAN, *buf* is the name of an array of 11 elements of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **STAT** system routine. In these examples, information about the file specified by the *path* parameter ("blue") is returned in the *buf* parameter ("yellow"). The value of the file mode for file /usr/include/aildefs.inc is the value printed out.

Pascal

```
procedure stat1;

type
  %include /usr/include/ailtypes.inc
var
  red : integer;
  yellow : statptr;
  blue : st80;

%include /usr/include/aildefs.inc

begin
  new (yellow);
  blue := '/usr/include/aildefs.inc';
  red := p_stat (blue, yellow);
  writeln (yellow@.st_mode)
end;
```

FORTRAN

```
      SUBROUTINE STAT1
      INTEGER STAT, RED, YELLOW(11)
      CHARACTER*80 BLUE
      BLUE = '/usr/include/aildef.inc'
      RED = STAT (BLUE, YELLOW)
      PRINT *, YELLOW(3)
      END
```

SYNC
— update a file system

Description

The **SYNC** system call writes modified information in core memory to disk, including modified super-blocks, i-nodes, and delayed block I/O.

Syntax

Pascal

```
p_sync;
```

FORTRAN

```
SYNC ()
```

Parameters

This system call has no parameters.

Return Values

The write operation may be scheduled but is not necessarily complete upon return from the **SYNC** call, and no value is returned.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SYNC** system routine. In these examples, all information in memory that should be on disk is written to disk.

Pascal

```
procedure sync1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;
```

```
%include /usr/include/aildefs.inc

begin
  blue := p_sync
end;
```

FORTRAN

```
      SUBROUTINE SYNC1
      INTEGER SYNC, BLUE
      BLUE = SYNC ()
      END
```

UMASK
— get and set file-creation-mode mask

Description

The **UMASK** system call sets a mask that is used whenever a file is created by a **CREAT** or **MKNOD** call. The access mode of the newly created file (see **CHMOD** on page 6-3) is set to the value of *cmask*. Only the low-order nine bits of the mask (the protection bits) participate.

Syntax

Pascal

```
p_umask (cmask);
```

FORTRAN

```
UMASK (CMASK)
```

Parameters

cmask
is the boolean complement of the new file's access mode.

- In Pascal, *cmask* is of type integer.
- In FORTRAN, *cmask* is of type INTEGER.

Return Values

The previous value of the mask is returned upon successful completion of the call. The initial value of the mask is 0 (zero), specifying "no restrictions."

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **UMASK** system routine with the value of the *cmask* parameter ("red") equal to 0 (zero). This value specifies the elimination of all restrictions on the file-creation mode. The value printed out is the previous value of the mask.

Pascal

```
procedure umask1;

type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;

%include /usr/include/aildefs.inc

begin
  red := 0;
  blue := p_umask (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE UMASK1
      INTEGER UMASK, BLUE, RED
      RED = 0
      BLUE = UMASK (RED)
      PRINT *, BLUE
      END
```

UNLINK
— delete a directory entry

Description

The **UNLINK** system call deletes the entry to a specified file from the directory in which it appears.

Note: Only the super-user can issue this call.

Syntax

Pascal

```
p_unlink (path);
```

FORTRAN

```
UNLINK (PATH)
```

Parameters

path
is the name of the file to be deleted.

- In Pascal, *path* is a string variable or constant of type st80,
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type INTEGER.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **UNLINK** system routine, which in these examples removes the directory entry specified in the *path* parameter ("blue"), assuming that file /tmp/xxx exists.

Pascal

```
procedure unlink1;

type
  %include /usr/include/ailtypes.inc
var
  yellow : integer;
  blue : st80;

%include /usr/include/aildefs.inc

begin
  blue := '/tmp/xxx';
  yellow := p_unlink (blue);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE UNLINK1
      INTEGER UNLINK, YELLOW
      CHARACTER*80 BLUE
      BLUE = '/tmp/xxx '
      YELLOW = UNLINK (BLUE)
      PRINT *, YELLOW
      END
```

USTAT
— get file-system information

Description

The **USTAT** system call retrieves and stores information about a mounted file system.

Syntax

Pascal

```
p_ustat (dev, buf);
```

FORTRAN

```
USTAT (DEV, ABUF, BBUF)
```

Parameters

dev

is the ID of the device corresponding to the element *strdev* of the data structure returned by **STAT**.

- In Pascal, *dev* is of type integer.
- In FORTRAN, *dev* is of type INTEGER.

buf

is the pointer to the data structure that holds the retrieved information.

- In Pascal, *buf* if of type ustatptr.
- In FORTRAN, *buf* is divided into two parameters:
  - *abuf* is an array(2) of type INTEGER.
  - *bbuf* is an array(2,6) of type CHARACTER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **USTAT** system routine. In these examples, information about the device specified by the *dev* parameter ("blue") is returned in the *buf* parameter ("yellow"). The value assigned to *dev*(1) specifies /dev/hd1. Normally this parameter value is obtained from a field of the information returned by a **STAT** call.

Pascal

```
procedure ustat1;

type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;
  yellow : ustatptr;

%include /usr/include/aildefs.inc

begin
  new (yellow);
  blue := 1;
  red := p_ustat (blue, yellow);
  writeln (red)
end;
```

FORTRAN

```
SUBROUTINE USTAT1
INTEGER USTAT, BLUE, GREEN(2), RED
CHARACTER YELLOW(2,6)
BLUE = 1
RED = USTAT (BLUE, GREEN, YELLOW)
PRINT *, RED
END
```

UTIME
— set the file times

Description

The **UTIME** system call sets the access and modification times of a specified file. The 'i-node changed' time of the file is set to the current time.

Note: Only the file owner and the super-user may issue this call.

Syntax

Pascal

```
p_utime (path, times);
```

FORTRAN

```
UTIME (PATH, TIMES)
```

Parameters

path
is the name of the file whose times are to be set.

- In Pascal, *path* is a string variable or a constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

times
is a pointer to a two-element array. The first element holds the new accessed time. The second element holds the new updated time.

- In Pascal, *times* is of type utimptr.
- In FORTRAN, *times* is the name of an array consisting of two elements of type INTEGER.

Note: If *times* is given the value nil in Pascal or -1 in FORTRAN, the access and modification *times* of the file in *path* are set equal to the current time.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **UTIME** system routine. In these examples, the access and modification times of the file specified by the *path* parameter ("blue") are set to the current time.

Pascal

```
procedure utime1;

type
  %include /usr/include/ailtypes.inc
var
  red : integer;
  blue : st80;
  yellow : utimptr;

%include /usr/include/aildefs.inc

begin
  blue := '/usr/include/ailtypes.inc';
  yellow := nil;
  red := p_utime (blue, yellow);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE UTIME1
      CHARACTER*80 BLUE
      INTEGER UTIME, RED, YELLOW(2)
      BLUE = '/usr/include/ailtypes.inc '
      YELLOW(1) = -1
      RED = UTIME (BLUE, YELLOW)
      PRINT *, RED
      END
```

Chapter 7. Signals

Signals provide a simple method of communication between two processes. One process can inform another process of status conditions or of the occurrence of an event. The first four system calls listed below can be used for standard signal processing. The remaining five provide several useful features for enhanced signal processing.

Call	Function
ALARM	schedules an alarm signal.
KILL	sends a signal to a process.
PAUSE	suspends the calling process until a signal is received.
SIGNAL	specifies the process response to a signal.
SIGBLK	blocks one or more signals.
SIGMSK	sets the signal mask of the current process.
SIGPAS	releases a blocked signal and waits for an interrupt.
SIGSTK	specifies an alternate stack for processing signals.
SIGVEC	selects signal-handling facitities.

Note: The **SIGSTK** and **SIGVEC** calls are not available in FORTRAN.

ALARM
— schedule an alarm signal

Description

The **ALARM** system call sends a signal (see page 7-12) to the calling process in a specified number of seconds. In effect, it sets an "alarm" clock. Unless caught or ignored, the signal terminates the calling process.

Syntax

Pascal

```
p_alarm (sec);
```

FORTRAN

```
ALARM (SEC)
```

Parameters

sec
is the number of seconds before the alarm signal is sent to the calling process (see **Notes** at the end of this section).

- In Pascal, *sec* is of type usign.
- In FORTRAN, *sec* is of type INTEGER.

Return Values

The return value of this call is the amount of clock time remaining from the previous **ALARM** call. The return value is the amount of time that previously remained on the alarm clock of the calling process before it is reset to the new time (see **Notes**).

- In Pascal, the return value is of type usign.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **ALARM** system routine, which in these examples instructs the alarm clock to signal the calling process after 100 seconds have elapsed.

Pascal

```
procedure alarm1;

type
  %include /usr/include/ailtypes.inc
var
  red, blue : usign;

%include /usr/include/aildefs.inc

begin
  red := 100;
  blue := p_alarm (red);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE ALARM1
      INTEGER ALARM, RED, BLUE
      RED = 100
      BLUE = ALARM (RED)
      PRINT *, BLUE
      END
```

Notes

Because Pascal and FORTRAN lack the facilities for handling unsigned four-byte integers, the programmer must convert parameter values of type usign that fall in the range

2 147 483 648 through 4 294 067 295

To use a parameter value in this range, subtract 4 294 067 296 from that value before issuing the call (the result will always be negative).

KILL
— send a signal to a process

Description

The **KILL** system call sends a specified signal to a specified process. The process receiving the signal is usually terminated as a result (see **SIGNAL** on page 7-12).

Note: Only the super-user may issue this call if the sending and receiving processes have different effective user ID's.

Syntax

Pascal

```
p_kill (pid, sig);
```

FORTRAN

```
KILL (PID, SIG)
```

Parameters

pid

is the ID of the process to which a signal is to be sent.

- In Pascal, *pid* is of type integer.
- In FORTRAN, *pid* is of type INTEGER.

sig

is the signal to be sent to the specified process. A process may send signals to itself.

- In Pascal, *sig* is of type integer.
- In FORTRAN, *sig* is of type INTEGER.

Return Values

The value 0 (zero) is returned if the specified process is terminated; the value -1 is returned if the receiving process does not have the same effective user ID as the sending process and the user is not the super-user. The value -1 is also returned if the specified process does not exist.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **KILL** system routine, which in these examples verifies the existence of the "special" root process, number 0.

Pascal

```
procedure kill1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;

%include /usr/include/aildefs.inc

begin
  blue := p_kill (0, 0);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE KILL1
      INTEGER KILL, BLUE
      BLUE = KILL (0, 0)
      PRINT *, BLUE
      END
```

PAUSE
— wait for a signal

Description

The **PAUSE** system call suspends the execution of a process until it receives a signal.

Syntax

Pascal

```
p_pause;
```

FORTRAN

```
PAUSE ()
```

Parameters

This system call has no parameters.

Return Values

There is no return value from a successful completion of **PAUSE**. The value -1 is returned and an error code is set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page call the **PAUSE** system routine, which suspends the calling process until the signal from the **ALARM** call is received.

Pascal

```
procedure pause1;

type
  %include /usr/include/ailtypes.inc
var
  blue : integer;
  red, green : usign;

%include /usr/include/aildefs.inc

begin
  red := 20;
  green := p_alarm (red);
  writeln (green);
  blue := p_pause
end;
```

FORTRAN

```
      SUBROUTINE PAUSE1
      INTEGER PAUSE, ALARM, BLUE, GREEN, RED
      RED = 20
      GREEN = ALARM (RED)
      PRINT *, GREEN
      BLUE = PAUSE ()
      END
```

SIGBLK
— block one or more signals

Description

The **SIGBLK** system call blocks one or more specified signals until a subsequent **SIGMSK** "unblocks" them (see page 7-12 for a complete list of signals).

Syntax

Pascal

```
p_sigblk (mask);
```

FORTRAN

```
SIGBLK (MASK)
```

Parameters

mask

specifies the signal(s) to be blocked by logically ORing the parameter value with the previous signal mask of the calling process.

Note: To set the mask value, use a number equal to 2 (two) raised to the (signal-number – 1) power. For example, the mask value that will block **SIGNAL** 31 is 2^{30} (see page 7-10).

- In Pascal, *mask* is of type integer.
- In FORTRAN, *mask* is of type INTEGER.

Return Values

The value that the signal mask had prior to the **SIGBLK** call is returned upon successful completion of the call.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SIGBLK** system routine, which in these examples blocks interrupt signals and illegal instruction signals that may be sent to the calling process. The return value printed out is equal to 2 (the previous masked blocked signal value) after the second call.

Pascal

```
procedure sigblk1;

type
  %include /usr/include/ailtypes.inc
var
  red, blue : integer;

%include /usr/include/aildefs.inc

begin
  red := p_sigblk (2);
  blue := p_sigblk (4);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE SIGBLK1
      INTEGER SIGBLK, BLUE, RED
      RED = SIGBLK (2)
      BLUE = SIGBLK (4)
      PRINT *, BLUE
      END
```

Notes

In the AIX Operating System, **SIGBLK** is named **sigblock**.

SIGMSK
— set the signal mask of the current process

Description

The **SIGMSK** system call sets the signal mask of the current process to a particular value, thereby specifying which signal will be blocked from receiving (that is, which signal the calling process will block).

Syntax

Pascal

```
p_sigmsk (mask);
```

FORTRAN

```
SIGMSK (MASK)
```

Parameters

mask
specifies the signal(s) to be blocked.

Note: To set the mask, use a number equal to 2 (two) raised to the (signal-number – 1) power. For example, the mask value that will block **SIGNAL** 31 is 2^{30}.

- In Pascal, *mask* is of type integer.
- In FORTRAN, *mask* is of type INTEGER.

Return Values

The value that the signal mask had before **SIGBLK** was called is returned on successful completion of the **SIGMSK** call.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SIGMSK** system routine, which in these examples blocks signal 14 (alarm clock). The call to **SIGBLK** returns the previous mask value, which should be what it has just been set to (8192). This mask value is also printed out ("orange").

Pascal

```
procedure sigmsk1;

type
  %include /usr/include/ailtypes.inc
var
  blue, orange, red : integer;

%include /usr/include/aildefs.inc

begin
  blue := 8192;
  red := p_sigmsk (blue);
  writeln (red);
  orange := p_sigblk (0);
  writeln (orange)
end;
```

FORTRAN

```
      SUBROUTINE SIGMSK1
      INTEGER SIGMSK, SIGBLK, BLUE, ORANGE, RED
      BLUE = 8192
      RED = SIGMSK (BLUE)
      PRINT *, RED
      ORANGE = SIGBLK (0)
      PRINT *, ORANGE
      END
```

Notes

In the AIX Operating System, **SIGMSK** is named **sigsetmask**.

SIGNAL
— specify the process response to a signal

Description

The **SIGNAL** system call sets the calling process to respond in one of three ways to the receipt of a signal:

- "catch" the signal;
- ignore the signal; or
- terminate its own execution (EXIT). Termination is the default event.

The signals that can be specified in a **SIGNAL** call are listed in the descriptions of the *sig* and *action* parameters.

Syntax

Pascal

```
p_signal (sig, action, func);
```

FORTRAN

```
SIGNAL (SIG, ACTION, FUNC)
```

Parameters

sig

is a number that specifies a particular signal. If a repeated signal arrives before the last one can be reset, it will not be caught (see **Notes**, item 2).

The signals that can be specified in a **SIGNAL** call are listed on the next page and are defined in the Pascal and FORTRAN constants include files.

Signal	Signal Number	Event
SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQT	3*	Quit
SIGILL	4*	Illegal instruction (not reset when caught)
SIGTRP	5*	Trace trap (not reset when caught)
SIGIOT	6*	Abort process
SIGDGR	7**	System is likely to crash soon
SIGFPE	8*	Floating-point exception, arithmetic exception, or divide by zero
SIGKIL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSGV	11*	Segmentation violation
SIGSYS	12*	Bad parameter

* This signal is a special case; see SIGNAL in *AIX Operating System Technical Reference*.

** This signal receives special handling; see SIGNAL in *AIX Operating System Technical Reference*.

Signal	Signal Number	Event
SIGPIP	13	Write on pipe when there is no process to read it
SIGALM	14	Alarm clock
SIGTRM	15	Software termination signal
SIGU1	16	User-defined signal 1
SIGU2	17	User-defined signal 2
SIGCLD	18**	Death of a child process
SIGPWR	19**	Power fail restart (not reset when caught)
SIGAIO	25	Basic LAN signal for asynchronous I/O
SIGPTY	26	PTY device driver read/write availability
SIGIO	27	I/O intervention required
SIGGNT	28+	HFT monitor access wanted
SIGRTC	29+	HFT monitor access should be relinquished
SIGSND	30+	An HFT sound control has completed execution
SIGMSG	31+	Message entered in a queue

+ For more information on this signal, see **hft** in *AIX Operating System Technical Reference*.

- In Pascal, *sig* is of type integer.
- In FORTRAN, *sig* is of type INTEGER.

action

specifies the action to be taken when one of the following signals is specified; SIGDFL, SIGIGN, or SIGFNC.

Parameter Value	Action
SIGDFL	Default action: Upon receipt of a signal, the receiving process is to be terminated.
SIGIGN	Ignore signal: The signal (SIG) is ignored by the receiving process.
SIGFNC	(function address) Catch signal: If the value of *action* is a function address, then upon receipt of the signal, the receiving process is to execute the signal-catching function specified by the *func* parameter.

- In Pascal, *action* is of type integer.
- In FORTRAN, *action* is of type INTEGER.

func

is used when a signal is to be caught and *action* is set equal to SIGFNC. This parameter directs the receiving process of the signal to execute the function specified. The *func* parameter is given the value nil in Pascal and 0 (zero) in FORTRAN if the value of *action* is SIGDFL or SIGIGN.

When calling a function from Pascal or FORTRAN, the function name should be the parameter.

- In Pascal, *func* is a function name.
- In FORTRAN, *func* is a function name.

Return Values

The previous value of *action* is returned for the specified *sig* upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutines shown here call the **SIGNAL** system routine. In this example, *sig* is assigned a value of 2 (SIGINT, interrupt signal). The *action* parameter is given the prescribed action SIGIGN, which causes the process to ignore the interrupt signal (that is, it does not terminate). The *func* parameter is sent as nil since no function address is needed in this instance.

Pascal

```
procedure signal1;

type
  %include /usr/include/ailtypes.inc
var
  blue, red, yellow : integer;

%include /usr/include/aildefs.inc

begin
  blue := 1;
  red := 2;
  yellow := p_signal (red, blue, nil);
  writeln (yellow)
end;
```

FORTRAN

```
      SUBROUTINE SIGNAL1
      INTEGER BLUE, RED, YELLOW, SIGNAL
      BLUE = 1
      RED = 2
      YELLOW = SIGNAL (RED, BLUE, 0)
      PRINT *, YELLOW
      END
```

Notes

1. The SIGKIL signal cannot be caught and it cannot be ignored.

2. The **SIGVEC** system call provides an enhanced signal-handling capacity that avoids this difficulty (see page 7-22).

SIGPAS
— release a blocked signal and wait for an interrupt

Description

The **SIGPAS** system call resets the signal mask of the calling process and causes the calling process to wait for a signal to arrive. The arrival of the signal terminates the call and restores the signal mask to its previous value. Upon completion of the call, the signal mask is restored to its previous value.

Syntax

Pascal

```
p_sigpas (sigmsk);
```

FORTRAN

```
SIGPAS (SIGMSK)
```

Parameters

sigmsk
is the value to which the signal mask of the calling process is set when the call is issued.

- In Pascal, *sigmsk* is of type integer.
- In FORTRAN, *sigmsk* is of type INTEGER.

Return Values

If the signal is caught by the calling process and control is returned from the signal handler, the calling process resumes execution after the **SIGPAS** system call, which always returns the value -1 and sets an error code in *errno*.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SIGPAS** system routine. In these examples, the first call is to the **ALARM** system routine, which sends a signal to the calling process after 10 seconds. The call to **SIGPAS** sets the signal mask to the value of the *sigmsk* parameter ("blue") to block interrupts.

Pascal

```
procedure sigpas1;

type
  %include /usr/include/ailtypes.inc
var
  blue, green, orange, red : integer;

%include /usr/include/aildefs.inc

begin
  orange := 10;
  green := p_alarm (orange);
  blue := 2;
  red := p_sigpas (blue)
end;
```

FORTRAN

```
      SUBROUTINE SIGPAS1
      INTEGER SIGPAS, BLUE, GREEN, ORANGE, RED
      ORANGE = 10
      GREEN = ALARM (ORANGE)
      BLUE = 2
      RED = SIGPAS (BLUE)
      END
```

Notes

In the AIX Operating System, **SIGPAS** is named **sigpause**.

SIGSTK
—define an alternate stack

Description

The **SIGSTK** system call defines an alternate stack on which signals are processed.

Warning: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results may occur.

Syntax

Pascal

```
p_sigstk (instack, outstack);
```

FORTRAN

```
Not available
```

Parameters

instack
points to a signal-stack data structure if the parameter value is **not** nil. If the parameter value **is** nil, then the signal-stack state is not set.

- In Pascal, *instack* is of type stackptr.

outstack
points to a signal-stack data structure if the parameter value is **not** nil. If the parameter value **is** nil, the previous signal-stack state is not reported.

- In Pascal, *outstack* is of type stackptr.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.

Examples

The Pascal procedure shown here calls the **SIGSTK** system routine. In this example the values being passed to **SIGSTK** are the *instack* ("yellow") and *outstack* ("green") parameters. The example merely shows the proper call: it neither sets a new stack nor stores the old (both parameters are set to nil).

Pascal

```
procedure sigstk1;

type
  %include /usr/include/ailtypes.inc
var
  blue, red : integer;
  green, yellow : stackptr;

%include /usr/include/aildefs.inc

begin
   new (yellow);
  new (green);
   new (yellow@.ss_sp);
  new (green@.ss_sp);
  yellow@.ss_sp := nil;
  green@.ss_sp := nil;
  red := p_sigstk (green, yellow);
  writeln (red)
end;
```

Notes

In the AIX Operating System, **SIGSTK** is named **sigstack**.

SIGVEC
— select signal-handling facilities

Description

The **SIGVEC** system call allows the user to select standard or enhanced signal-handling facilities. Like the **SIGNAL** call, it specifies the action to be taken on receipt of a given signal.

Warning: The **SIGVEC** call does not check the validity of the sv__handler pointer. If this pointer is pointing outside the address space of the process, a memory-fault message is returned to the process when the system attempts to use the signal handler.

Syntax

Pascal

```
p_sigvec (sig, code, invec, outvec);
```

FORTRAN

```
Not available
```

Parameters

sig

is the identifying number of a signal (See page 7-12 for a complete list of signals.).

- *sig* is of type integer.

code

specifies one of the following actions taken on the signal specified by *sig* (as defined in the Pascal constants include file).

SIGDFL	the *sig* default actions are reinstated.
SIGIGN	the signal is ignored.
SIGADDR	control is transferred to the address specified in *invec*.sv__handler.

- *code* is of type integer.

invec

specifies a handler routine and mask for use in delivering a signal when the parameter value is **not** nil. When the parameter value **is** nil, the signal-handler information is not set. The value of the sv__onstak field of the *invec* record specifies one of three options:

0 the enhanced signal and the process signal on the process stack are used.

1 the enhanced signal and the process signal on a separate stack are used.

2 standard signal processing is used.

- *invec* is of type sigvecptr.

outvec

points to a record where the previous handling information for the signal in the structure is stored, when it is **not** nil. Information for the signal is stored in the **SIGVEC** data structure pointed to by *outvec*. If the value of the *outvec* parameter is nil, the previous signal-handler information is not reported.

- *outvec* is of type sigvecptr.

Return Values

There is no return value from a successful **SIGVEC** call.

Examples

The Pascal procedure shown on the next page calls the **SIGVEC** system routine. In these examples, the value passed to **SIGVEC** by the parameter *sig* ("yellow"), specifies signal (2) and the *invec* and *outvec* parameters ("blue" and "red", respectively). The default action is specified by the variable "orange"; the *invec* and *outvec* parameters are set equal to 'nil' since they are not necessary for this action.

Pascal

```
procedure sigvec1;

const
  %include usr/include/pconst.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, red : sigvecptr;
  green, orange, yellow : integer;

%include /usr/include/aildefs.inc

begin
  yellow := 2;
  orange := SIGDFL;
  new (blue);
  new (red);
  red := nil;
  blue@.sv_handler := nil;
  blue@.sv_mask := 0;
  blue@.sv_onstack := 0;
  green := p_sigvec (yellow, orange, blue, red)
end;
```

Chapter 8. Semaphores

The following system calls can be used to create semaphores, which are similar to signals but have additional features that make them more useful for interprocess communication.

Call	Function
SEMCTL	invokes semaphore-control operations.
SEMGET	gets or creates a semaphore-set ID.
SEMOP	performs semaphore operations.

SEMCTL
— invoke semaphore-control operations

Description

The **SEMCTL** system call invokes a variety of semaphore-control operations, most of which involve getting and setting the values of a data structure containing information about a set of semaphores.

Syntax

Pascal

```
p_semctl (semid, semnum, cmd, arg);
```

FORTRAN

```
SEMCTL (SEMID, SEMNUM, CMD, ARG)
```

Parameters

semid
is the identifier of a semaphore set created by a previous **SEMGET** call (see page 8-9). The value of *semid* is returned by the **SEMGET** call.

- In Pascal, *semid* is of type integer.
- In FORTRAN, *semid* is of type INTEGER.

semnum
specifies the particular semaphore that will be affected by the control operation invoked by the call.

- In Pascal, *semnum* is of type integer.
- In FORTRAN, *semnum* is of type INTEGER.

cmd

specifies the control operation to be performed, which can be any of the options in the following list. These options are executed with respect to the semaphores specified by *semid* and *semnum*.

Note: Each constant is defined in the Pascal and FORTRAN constants include file (see Appendix B).

The fields referred to in the option descriptions below belong to the sem record (see Appendix C).

GETVAL returns the value of the semval field of the semaphore specified by *semid* and *semnum*.

SETVAL sets the value of the semval field of the semaphore set according to the array pointed to by the field arg.val.

GETPID returns the value of the sempid field of the semaphore specified by *semid* and *semnum*.

GTNCNT returns the value of the semncnt field of the semaphore specified by *semid* and *semnum*.

GTZCNT returns the value of the semzcnt field of the semaphore specified by *semid* and *semnum*.

The following *cmd* options return and set every semval field in the set of semaphores.

GETALL takes the values of the semval field of the semaphore specified by *semid* and *semnum* and stores them in the array pointed to by the field arg.arry.

SETALL sets semvals according to the array pointed to by arg.arry.

IPCSTT takes the current value of each field of the data structure associated with *semid* and stores it in the structure pointed to by the field arg.buf. In FORTRAN, information is stored in the first 14 ele-

ments of the field arg.arry (for further information see Table A on page 8-5).

IPCSET sets the value of the following fields of the data structure associated with *semid* to the corresponding values found in the structure pointed to by arg.buf.

- sem_perm.uid
- sem_perm.gid
- sem_perm.mode (low-order nine bits only)

In FORTRAN these fields are set according to elements 1, 2, and 5 of the field arg.arry.

Note: This option can be used only when the effective user ID is equal to the super-user ID or to the user ID.

IPCRMD removes the semaphore identifier and its associated data structure from the operating system.

Note: This option can be used only when the effective user ID is equal to the super-user ID or to the user ID.

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

arg

is a data structure determined by the *cmd* parameter. The *cmd* options return values to the Pascal record and the FORTRAN array as follows.

For *cmd* options GETVAL, SETVAL, GETPID, GTNCNT, and GTZCNT:

Pascal	FORTRAN	Description
arg.val	ARG(1)	The values of the sem record are set and returned here.

For *cmd* options GETALL and SETALL:

Pascal	FORTRAN	Description
arg.arry@[1]	ARG(1)	The values of the semary record are set and returned here.
. . .	. . .	
arg.arry@[1000]	ARG(1000)	

For *cmd* options IPCSTT and IPCSET:

Pascal	FORTRAN	Description
arg.buf@.sem__perm.uid	ARG(1)	owner's user ID
arg.buf@.sem__perm.gid	ARG(2)	owner's group ID
arg.buf@.sem__perm.cuid	ARG(3)	creator's user ID
arg.buf@.sem__perm.cgid	ARG(4)	creator's group ID
arg.buf@.sem__perm.mode	ARG(5)	access mode

Pascal	FORTRAN	Description
arg.buf@.sem__perm.seq	ARG(6)	lot-usage sequence number
arg.buf@.sem__perm.key	ARG(7)	key value
arg.buf@.sem__base@.semval	ARG(8)	operation permission structure
arg.buf@.sem__base@.sempid	ARG(9)	ID of last process that issued **SEMOP**
arg.buf@.sem__base@.semncnt	ARG(10)	number of processes awaiting semval > cval
arg.buf@.sem__base@.semzcnt	ARG(11)	number of processes awaiting semval = 0
arg.buf@.sem__nsems	ARG(12)	number of semaphores in a set
arg.buf@.semlcnt	ARG(13)	processes waiting on locked semaphore
arg.buf@.sem__otime	ARG(14)	time of last **SEMOP** call
arg.buf@.sem__ctime	ARG(15)	last time this structure was changed by a **SEMCTL** call

- In Pascal, *arg* is of type semrec.
- In FORTRAN, *arg* is a 1000-element array of type INTEGER.

Return Values

The value returned from a successful call varies with the *cmd* option specified.

GTNCNT	semncnt
GETPID	sempid
GETVAL	semval
GTZCNT	semzcnt
All Others	0

The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SEMCTL** system routine. In these examples, a semaphore identifier is retrieved by a call to **SEMGET** from the associated key parameter ("red") returned by a call to the **ftok** system subroutine. The call to **SEMCTL** stores the current value of each member of the data structure associated with the *semid* parameter ("green") in the structure yellow.buf (in Pascal) or YELLOW(1)..YELLOW(15) in FORTRAN.

Pascal

```
procedure semctl1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, pink, red, purple : integer;
  orange : st80;
  brown : char;
  yellow : semrec;

%include /usr/include/aildefs.inc

begin
   new (yellow.buf);
  brown := 'm';
  orange := '/tmp/junk';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_semget (red, 20, blue);
  pink := 20;
  purple := p_semctl (green, pink, 2, yellow);
  writeln (purple)
end;
```

FORTRAN

```
SUBROUTINE SEMCTL1
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER SEMCTL, FTOK, BLUE, GREEN, PINK
INTEGER PURPLE, RED, YELLOW(1000), SEMGET
CHARACTER BROWN, ORANGE*80
BROWN = 'm'
ORANGE = '/tmp/junk '
BLUE = IPCCRT + IRUSR
RED = FTOK (ORANGE, BROWN)
GREEN = SEMGET (RED, 20, BLUE)
PINK = 20
PURPLE = SEMCTL (GREEN, PINK, 2, YELLOW)
PRINT *, PURPLE
END
```

SEMGET
— get or create a semaphore-set ID

Description

The **SEMGET** system call returns a semaphore-set ID associated with the specified *key* parameter.

Syntax

Pascal

```
p_semget (key, nsems, semflg);
```

FORTRAN

```
SEMGET (KEY, NSEMS, SEMFLG)
```

Parameters

key
is a semaphore-set ID that has been assigned directly by the programmer or has been returned by the **ftok** system subroutine or similar algorithm.

- In Pascal, *key* is of type integer.
- In FORTRAN, *key* is of type INTEGER.

nsems
specifies the number of semaphores in a set.

- In Pascal, *nsems* is of type integer.
- In FORTRAN, *nsems* is of type INTEGER.

semflg
specifies one or more conditions (options) governing the creation of a semaphore-set data structure and the accessibility of the semaphore set. The parameter value is that of one of the following options or is

constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files.

IPCCRT creates a data structure if one does not exist.

IPCEXL causes **SEMGET** to fail if IPCCRT is also set and the data structure already exists.

IRUSR permits the process that owns the data structure to read it.

IWUSR permits the process that owns the data structure to modify it.

IRGRP permits the group associated with the data structure to read it.

IWGRP permits the group associated with the data structure to modify it.

IROTH permits others to read the data structure.

IWOTH permits others to modify the data structure.

- In Pascal, *semflg* is of type integer.
- In FORTRAN, *semflg* is of type INTEGER.

Return Values

A semaphore-set ID is returned upon successful completion of the call. The value -1 is returned and error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SEMGET** system routine, which in these examples returns a semaphore identifier associated with the key parameter ("red") returned by a call to the **ftok** system subroutine. This identifier is the value printed out.

Pascal

```
procedure semget1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, red : integer;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  brown := 'm';
  orange := '/tmp/junk';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_semget (red, 20, blue);
  writeln (green)
end;
```

FORTRAN

```
      SUBROUTINE SEMGET1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER SEMGET, FTOK, BLUE, GREEN, RED
      CHARACTER BROWN, ORANGE*80
      BROWN = 'm'
      ORANGE = '/tmp/junk '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = SEMGET (RED, 20, BLUE)
      PRINT *, GREEN
      END
```

SEMOP
— perform semaphore operations

Description

The **SEMOP** system call invokes a group of semaphore operations that are performed on a specified semaphore set.

Syntax

Pascal

```
p_semop (semid, sops, nsops);
```

FORTRAN

```
SEMOP (SEMID, SOPS, NSOPS)
```

Parameters

semid
is the ID of the semaphore set that is to be operated on.

- In Pascal, *semid* is of type integer.
- In FORTRAN, *semid* is of type INTEGER.

sops
is a pointer to an array of semaphore operation data structures. The breakdown of this parameter for each of the **n** semaphores is as follows:

Pascal	FORTRAN	Description
nsops[n]sem__num	NSOPS(n,1)	Semaphore number
nsops[n]sem__op	NSOPS(n,2)	Semaphore operation
nsops[n].sem__flg	NSOPS(n,3)	Operation flags

Each semaphore operation specified by sem__op (FORTRAN, NSOPS(n,2)) is performed on the corresponding semaphore specified by sem__num (FORTRAN, NSOPS(n,1)). The sem__flg (FORTRAN, NSOPS(n,3)) value can be 0, one of the following constants, or the value obtained from logically ORing (adding) the following constants defined in the Pascal and FORTRAN constants include files.

SEMNDO (SEM__UNDO)
SEMODR (SEM__ORDER)
IPCNWT (IPC__NOWAIT)

Note: For further information about these constants and the semaphore operations, see *AIX Operating System Technical Reference*.

- In Pascal, *sops* is of type semopary.
- In FORTRAN, *sops* is an array(1000,3) of type INTEGER.

nsops
specifies the number of semaphore operations to be performed. A semaphore set is limited to 1000 semaphores.

- In Pascal, *nsops* is of type integer.
- In FORTRAN, *nsops* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. In addition, each value of *sempid* for each semaphore in the array pointed to by *sops* is set to the process ID of the calling process. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SEMOP** system routine. In these examples, a semaphore identifier is retrieved by a call to **SEMGET** from the associated *key* parameter ("red") returned by a call to the **ftok** system subroutine. The call to **SEMGET** would typically be part of a program used between two processes using semaphores to buffer information. The call to **SEMOP** is used by the sending process to perform two semaphore operations. The first operation decrements a counter of empty buffer available upon sending information. The second operation increments a second counter of data packages that can be received by a second process.

Pascal

```
procedure semop1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, grey, pink, red : integer;
  orange : st80;
  brown : char;
  yellow : semopary;

%include /usr/include/aildefs.inc

begin
  brown := 'z';
  orange := '/tmp/junk';
  grey := IPCCRT + IRUSR + IWUSR;
  red := p_ftok (orange, brown);
  pink := p_semget (red, 2, grey);
  yellow[1].sem_num := 1;
  yellow[2].sem_num := 2;
  yellow[1].sem_op := -1;
  yellow[2].sem_op := 1;
  yellow[1].sem_flg := 0;
  yellow[2].sem_flg := 0;
  blue := p_semop (pink, yellow, 2);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE SEMOP1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER SEMOP, SEMGET, FTOK, BLUE
      INTEGER GREY, PINK, RED, YELLOW(1000,3)
      CHARACTER BROWN, ORANGE*80
      BROWN = 'z'
      ORANGE = '/tmp/junk '
      GREY = IPCCRT + IRUSR + IWUSR
      RED = FTOK (ORANGE, BROWN)
      PINK = SEMGET (RED, 2, GREY)
      YELLOW(1,1) = 1
      YELLOW(2,1) = 2
      YELLOW(1,2) = -1
      YELLOW(2,2) = 1
      YELLOW(1,3) = 0
      YELLOW(2,3) = 0
      BLUE = SEMOP (PINK, YELLOW, 2)
      PRINT *, BLUE
      END
```

Chapter 9. Messages

The following system calls can be used to pass information from one process to another and to create and use message queues. The information may be of any kind, including data generated by one process and used in another, and flags that indicate when events occur.

Call	Function
MSGCTL	invokes message-control operations.
MSGGET	gets or creates a message queue.
MSGRV, MSGXRV	read and store a message.
MSGSND	sends a message to a queue.

MSGCTL
— invoke message-control operations

Description

The **MSGCTL** system call invokes any of three message-control operations, including the storing and setting of the values in a specified message queue.

Syntax

Pascal

```
p_msgctl (msqid, cmd, buf);
```

FORTRAN

```
MSGCTL (MSQID, CMD, BUF)
```

Parameters

msqid
is the identifier of a message queue created by a previous **MSGGET** call (see page 9-6). The value of *msqid* is returned by **MSGGET**.

- In Pascal, *msqid* is of type integer.
- In FORTRAN, *msqid* is of type INTEGER.

cmd
specifies the operation to be performed, which can be any of the options in the following list.

Note: Each option number corresponds to a mnemonic (shown in parentheses) defined in the Pascal and FORTRAN constants include files.

0 (IPCRMD) removes the message-queue identifier and its associated data structure from the operating system and destroys the associated message.

1 (IPCSET) sets the value of the following fields and the data structure associated with *msqid* to the corresponding value found in the data structure pointed to by *buf*.

In Pascal these fields are:

- msg__perm.uid
- msg__perm.gid
- msg__perm.mode
- msg__qbytes

In FORTRAN the corresponding fields are:

- MSQID(1)
- MSQID(2)
- MSQID(5)
- MSQID(12)

Note: Only a process whose effective user ID is super-user can raise the value of msg__qbytes.

2 (IPCSTT) takes the current value of each field of the data structure associated with *msqid* and stores it in the structure pointed to by the *buf* parameter (see below).

Note: Options 0 and 1 can be used only when the effective user ID is equal to the super-user ID or to the value of msqid__ds@.msg__perm.uid in Pascal or MSQID(1) in FORTRAN.

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

buf

points to a record of type msqid__ds. The values stored or set in this record are the current values of the data structure associated with *msqid*.

- In Pascal, *buf* is of type mdsptr.
- In FORTRAN, *buf* is an array(17) of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **MSGCTL** system routine. The value of the first parameter of this call is the return value of **MSGGET**. (The value of the first parameter of **MSGGET** is the return value of the **ftok** system subroutine; see **Notes** at the end of this section.) The variable "pink" specifies the option that stores the values associated with the *msqid* parameter ("green") in the data structure provided ("yellow").

Pascal

```
procedure msgctl1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, blue, green, pink : integer;
  yellow : mdsptr;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc
```

```
begin
  new (yellow);
  brown := 'm';
  orange := '/usr/include/ailtypes.inc';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_msgget (red, blue);
  pink := 2;
  red := p_msgctl (green, pink, yellow);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE MSGCTL1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER FTOK, MSGGET, MSGCTL, RED, GREEN, BLUE, PINK
      INTEGER YELLOW(17)
      CHARACTER*80 ORANGE
      CHARACTER BROWN
      BROWN = 'm'
      ORANGE = '/usr/include/ailtypes.inc '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = MSGGET (RED, BLUE)
      PINK = 2
      RED = MSGCTL (GREEN, PINK, YELLOW)
      PRINT *, RED
      END
```

MSGGET
— get or create a message queue

Description

The **MSGGET** system call gets a specified message queue identifier associated with the specified key parameter. **MSGGET** can also create the identifier and message queue if they do not already exist.

Syntax

Pascal

```
p_msgget (key, msgflg);
```

FORTRAN

```
MSGGET (KEY, MSGFLG)
```

Parameters

key
: determines which identifier and associated data structure to use. The *key* parameter may be equal to **0** (IPCPVT); or *key* can be an IPC key constructed by a call to the **ftok** system subroutine.

- In Pascal, *key* is of type integer.
- In FORTRAN, *key* is of type INTEGER.

msgflg
: specifies a set of conditions (options) governing the creation of the message-queue data structure and the accessibility of the message queue. The parameter value is that of one of the following options or is constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files.

IPCCRT creates the message-queue data structure when it does not exist.

IPCEXL causes **MSGGET** to fail when IPCCRT is set and the message-queue data structure exists.

IRUSR permits the process that owns the message-queue data structure to read it.

IWUSR permits the process that owns the message-queue data structure to modify it.

IRGRP permits the group associated with the message-queue data structure to read it.

IWGRP permits the group associated with the message-queue data structure to modify it.

IROTH permits others to read the message-queue data structure.

IWOTH permits others to modify the message-queue data structure.

- In Pascal, *msgflg* is of type integer.
- In FORTRAN, *msgflg* is of type INTEGER.

Return Values

A message-queue identifier is returned upon successful completion of the call, and the data structure (msqid__ds; see Appendix C) associated with the new identifier is initialized. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Example

The Pascal procedure and FORTRAN subroutine shown here call the **MSGGET** system routine. (The value of the first parameter of the call is the return value of the **ftok** system subroutine.0 The value assigned to the second parameter ("blue") specifies the creation of a message queue for the process (if one does not already exist) and gives the user read access to it.

Pascal

```
procedure msgget1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  red, green, blue : integer;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  brown := 'm';
  orange := '/usr/include/ailtypes.inc';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_msgget (red, blue);
  writeln (green)
end;
```

FORTRAN

```
      SUBROUTINE MSGGET1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER MSGGET, FTOK, BLUE, GREEN, RED
      CHARACTER*80 ORANGE
      CHARACTER BROWN
      BROWN = 'm'
      ORANGE = '/usr/include/ailtypes.inc '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = MSGGET (RED, BLUE)
      PRINT *, GREEN
      END
```

MSGRV, MSGXRV
— read and store a message

Description

Both of the **MSG** system calls read a message from a specified queue and place it in a structure specified in the call.

In addition, **MSGXRV** will return the following items of information:

- the time the message was sent.
- the sender's effective user ID.
- the sender's effective group ID.
- the sender's node ID.
- the sender's process ID.

Syntax

Pascal

```
p_msgrv (msqid, msgp, msgsz, msgtyp, msgflg);

p_msgxrv (msqid, msgpt, msgsz, msgtyp, msgflg);
```

FORTRAN

```
MSGRV (MSQID, MSGP1, MSGP2, MSGSZ, MSGTYP, MSGFLG)

MSGXRV (MSQID, MSGPT1, MSGPT2, MSGSZ, MSGTYP, MSGFLG)
```

Parameters

msqid
is a message-queue identifier containing the message to be read.

- In Pascal, *msqid* is of type integer.
- In FORTRAN, *msqid* is of type INTEGER.

msgp
points to the record msgbuf, in which a type identifier and the message will be stored. This message is read from the queue specified by *msqid*. The *msgp* parameter is used only in the **MSGRV** call.

- In Pascal, *msgp* is of type mbufptr.
- In FORTRAN, *msgp* is sent as two parameters:
  - *msgp1* is of type INTEGER.
  - *msgp2* is of type CHARACTER*80.

msgpt
points to the extended message receive buffer (msgxbuf), in which the message time, sender information, type identifier, and message will be stored. This message is read from the queue specified by *msqid*. The *msgpt* parameter is used only in the **MSGXRV** call.

- In Pascal, *msgpt* is of type msgxptr.
- In FORTRAN, *msgpt* is sent as two parameters:
  - *msgpt1* is an array(6) of type INTEGER.
  - *msgpt2* is of type CHARACTER*80.

msgsz
is a constant or variable that specifies the length of the message in bytes. The maximum size of *msgsz* is 80 characters.

- In Pascal, *msgsz* is of type integer.
- In FORTRAN, *msgsz* is of type INTEGER.

msgtyp
is a constant or variable that specifies the type of the message to be read.

- In Pascal, *msgtyp* is of type integer.
- In FORTRAN, *msgtyp* is of type INTEGER.

msgflg
specifies the operation to be performed when the desired message is in the queue and when it is not. The value assigned to *msgflg* is that of one or more of the following:

IPCNER
truncates the message when it is longer than the number of bytes specified by *msgsz*.

IPCNWT
specifies the operation to be performed when the desired message is not in the queue.

- In Pascal, *msgflg* is of type integer.
- In FORTRAN, *msgflg* is of type INTEGER.

Return Values

A value equal to the number of bytes stored in mtext (of msgbuf or msgxbuf) is returned upon successful completion of a call, and the data structure associated with the message-queue identifier is modified as follows:

- msg__qnum is decremented by 1.
- msg__lpid is set equal to the process ID of the calling process.
- msg__rtime is set equal to the current time.

The value -1 is returned and an error code is set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **MSGRV** system routine. The value of the first parameter of this call is the return value of **MSGGET**. (The value of the first parameter of **MSGGET** is the return value of the **ftok** system subroutine; see **Notes** at the end of this section.) The variable "orange" specifies the maximum length of the message. The value printed out is the number of bytes received from a message.

Pascal

```
procedure msgrv1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, grey, orange, pink, purple, red : integer;
  yellow : mbufptr;
  white : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  new (yellow);
  brown := 'w';
  white := '/usr/include/ailtypes.inc';
  blue := 0;
  green := IPCNER;
  orange := 50;
  pink := IPCCRT + IRUSR;
  purple := p_ftok (white, brown);
  red := p_msgget (purple, pink);
  grey := p_msgrv (red, yellow, orange, blue, green);
  writeln (grey)
end;
```

FORTRAN

```
SUBROUTINE MSGRV1
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER MSGRV, MSGGET, FTOK, GREY, VIOLET
INTEGER BLUE, GREEN, ORANGE, PINK, PURPLE, RED
CHARACTER*80 WHITE, YELLOW
CHARACTER BROWN
BROWN = 'w'
WHITE = '/usr/include/ailtypes.inc '
BLUE = 0
GREEN = IPCNER
ORANGE = 50
PINK = IPCCRT + IRUSR
PURPLE = FTOK (WHITE, BROWN)
RED = MSGGET (PURPLE, PINK)
GREY = MSGRV (RED, VIOLET, YELLOW, ORANGE, BLUE, GREEN)
PRINT *, GREY
END
```

Notes

In the AIX Operating System, **MSGRV** and **MSGXRV** are named **msgrcv** and **msgxrcv**, respectively.

MSGSND
— send a message to a queue

Description

The **MSGSND** system call sends a message to a specified queue.

Syntax

Pascal

```
p_msgsnd (msqid, msgp, msgsz, msgtyp, msgflg);
```

FORTRAN

```
MSGSND (MSQID, MSGP1, MSGP2, MSGSZ, MSGFLG)
```

Parameters

msqid
is a message-queue identifier to which a message is to be sent.

- In Pascal, *msqid* is of type integer.
- In FORTRAN, *msqid* is of type INTEGER.

msgp
is the pointer to the record msgbuf, which contains the message to be sent.

- In Pascal, *msgp* is of type mbufptr.
- In FORTRAN, *msgp* is sent as two parameters:
  - *msgp1* is of type INTEGER.
  - *msgp2* is of type CHARACTER*80.

msgsz
is a constant or variable that specifies the length of the message in bytes. The maximum size of *msgsz* is 80 bytes.

- In Pascal, *msgsz* is of type integer.
- In FORTRAN, *msgsz* is of type INTEGER.

msgflg
specifies the action taken when either of the following conditions prevents the message from being sent:

- the number of bytes already in the queue is equal to the number specified by msg__qbytes.
- the total number of messages in all queues in the system is equal to the system-imposed limit.
- In Pascal, *msgflg* is of type integer.
- In FORTRAN, *msgflg* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call, and the data structure associated with the message-queue identifier is modified as follows:

- msg__qnum is incremented by 1.
- msg__lspid is set equal to the process ID of the calling process.
- msg__stime is set equal to the current time.

The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Example

The Pascal procedure and FORTRAN subroutine shown here call the **MSGSND** system routine. The value of the first parameter of this call is the return value of **MSGGET**. (The value of the first parameter of **MSGGET** is the return value of the **ftok** system subroutine; see **Notes** at the end of this section.) The variable "orange" specifies the length of the message.

Pascal

```
procedure msgsnd1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc

var
  blue, grey, orange, pink, purple, red : integer;
  yellow : mbufptr;
  white : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  new (yellow);
  brown := 'w';
  white := '/usr/include/ailtypes.inc';
  blue := IPCNWT;
  orange := 27;
  pink := IPCCRT + IRUSR + IWUSR;
  yellow@.mtype := 1;
  yellow@.mtext := 'This is 1 test for messages';
  purple := p_ftok (white, brown);
  red := p_msgget (purple, pink);
  grey := p_msgsnd (red, yellow, orange, blue);
  writeln (grey)
end;
```

FORTRAN

```
SUBROUTINE MSGSND1
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER MSGSND, MSGGET, FTOK, GREY, PURPLE
INTEGER BLUE, ORANGE, PINK, RED, YELLOW
CHARACTER*80 WHITE, VIOLET
CHARACTER BROWN
BROWN = 'w'
WHITE = '/usr/include/ailtypes.inc '
BLUE = IPCNWT
ORANGE = 27
PINK = IPCCRT + IRUSR + IWUSR
YELLOW = 1
VIOLET = 'This is 1 test for messages'
PURPLE = FTOK (WHITE, BROWN)
RED = MSGGET (PURPLE, PINK)
GREY = MSGSND (RED, YELLOW, VIOLET, ORANGE, BLUE)
PRINT *, GREY
END
```

Chapter 10. Shared Memory

The following system calls can be used to set aside an area of memory that cooperating processes can access.

Call	Function
SHMAT	attaches a shared-memory segment or mapped file to a process.
SHMCTL	invokes shared-memory-control operations.
SHMDT	detaches a shared-memory segment from a process.
SHMGET	gets a shared-memory segment identifier.

SHMAT
— attach a shared-memory segment or mapped file

Description

The **SHMAT** system call attaches one of the following to the address space of the calling process:

- a shared memory segment, or
- a mapped file associated with a shared-memory identifier (returned by **SHMGET**), or
- a file descriptor (returned by **OPEN**).

Syntax

Pascal

```
p_shmat (shmid, shmadr, shmflg);
```

FORTRAN

```
SHMAT (SHMID, SHMADR, SHMFLG)
```

Parameters

shmid
is either a shared-memory identifier returned by **SHMGET** or a file descriptor returned by **OPEN**.

- In Pascal, *shmid* is of type integer.
- In FORTRAN, *shmid* is of type INTEGER.

shmadr
determines the address to which the shared-memory segment is attached.

- In Pascal, *shmadr* is of type integer.
- In FORTRAN, *shmadr* is of type INTEGER.

shmflg

specifies a set of conditions governing the attachment of a shared-memory segment or a mapped file to an address space. The value assigned to *shmflg* is that of one or more of the options in the following list. These are defined in the Pascal and FORTRAN constants include files.

SHMRND rounds the address given by the *shmadr* parameter to the next lower segment boundary if necessary.

SHMMAP maps a file, instead of a shared-memory segment, onto the address space. The *shmid* parameter must specify an open-file descriptor and the file must be a regular file.

SHMRDO specifies read-only mode (the default is read-write mode).

SHMCPY Maps a file in copy-on-write mode.

Note: Either SHMRDO or SHMCPY may be specified, but not both.

- In Pascal, *shmflag* is of type integer.
- In FORTRAN, *shmflg* is of type INTEGER.

Return Values

The start address of the attached shared-memory segment or mapped file is returned on successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SHMAT** system routine. In these examples, the shared-memory identifier returned by a **SHMGET** call is used to specify the shared-memory segment that **SHMAT** attaches to the address of the calling process.

Pascal

```
procedure shmat1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, red : integer;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  brown := 'm';
  orange := '/tmp/junk';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_shmget (red, 512, blue);
  blue := p_shmat (green, 0, 0);
  writeln (blue)
end;
```

FORTRAN

```
      SUBROUTINE SHMAT1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER SHMAT, SHMGET, FTOK, BLUE, GREEN, RED
      CHARACTER BROWN, ORANGE*80
      BROWN = 'm'
      ORANGE = '/tmp/junk '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = SHMGET (RED, 512, BLUE)
      BLUE = SHMAT (GREEN, 0, 0)
      PRINT *, BLUE
      END
```

SHMCTL
— invoke shared-memory-control operations

Description

The **SHMCTL** system call invokes three shared-memory-control operations.

Syntax

Pascal

```
p_shmctl (shmid, cmd, buf);
```

FORTRAN

```
SHMCTL (SHMID, CMD, BUF)
```

Parameters

shmid
is a shared-memory-segment identifier returned by the **SHMGET** call.

- In Pascal, *shmid* is of type integer.
- In FORTRAN, *shmid* is of type INTEGER

cmd
specifies the control operation to be performed. These operations are defined in the Pascal and FORTRAN constants include files.

IPCRMD removes the shared-memory identifier specified by *shmid* from the system and erases the shared-memory segment and associated data structure.

Note: This option can be executed only by a process that has an effective user ID equal to that of the super-user or to the value of shm.perm.uid in the data structure.

IPCSET sets the value of the following members of the data structure associated with *shmid* to the corresponding value found in the structure pointed to by the *buf* parameter:

- shm.perm.uid
- shm.per.gid
- shm.perm.mode (low-order nine bits only)

Note: This *cmd* option can be executed only by a process that has an effective user ID equal to that of super-user or to the value of shm.perm.uid in the data structure associated with the *shmid* parameter.

IPCSTT places the current value of each member of the data structure associated with *shmid* in the structure pointed to by the *buf* parameter. The current process must have read permissions on this shared-memory segment or mapped file.

- In Pascal, *cmd* is of type integer.
- In FORTRAN, *cmd* is of type INTEGER.

buf

is a pointer to the data structure to be modified.

- In Pascal, *buf* is of type smdsptr.
- In FORTRAN, *buf* is an array(16) of type INTEGER.

The breakdown of the FORTRAN shared-memory data structure is as follows:

```
buf(1) = shperm.uid
buf(2) = shperm.gid
buf(3) = shperm.cuid
buf(4) = shperm.cgid
buf(5) = shperm.mode
buf(6) = shperm.seq
buf(7) = shperm.key
buf(8) = shsegsz
```

buf(9) = shsgid
buf(10) = shlpid
buf(11) = shcpid
buf(12) = shnattach
buf(13) = shcnattach
buf(14) = shatime
buf(15) = shdtime
buf(16) = shctime

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SHMCTL** system routine. In these examples, the *cmd* parameter ("pink") specifies an option that will place information about a shared-memory segment (identified by the *shmid* parameter, or "green") in the data structure pointed to by the *buf* parameter. In Pascal this structure is the record pointed to by the variable "yellow". In FORTRAN, "YELLOW" is an array. The value printed is the process user ID.

Pascal

```
procedure shmctl1;

const
  %include/usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, pink, red : integer;
  orange : st80;
  brown : char;
  yellow : smdsptr;

%include /usr/include/aildefs.inc
```

```
begin
   new (yellow);
  brown := 'm';
  orange := '/tmp/junk';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_shmget (red, 512, blue);
  pink := 2;
  red := p_shmctl (green, pink, yellow);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE SHMCTL1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER SHMCTL, SHMGET, FTOK, BLUE, GREEN, PINK, RED
      CHARACTER BROWN, ORANGE*80
      INTEGER YELLOW(16)
      BROWN = 'm'
      ORANGE = '/tmp/junk '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = SHMGET (RED, 512, blue)
      PINK = 2
      RED = SHMCTL (GREEN, PINK, YELLOW)
      PRINT *, RED
      END
```

SHMDT
— detach a shared-memory or mapped file segment

Description

The **SHMDT** system call detaches a shared-memory segment from the data segment of the calling process. Shared memory segments must be explicitly detached using **SHMDT**.

Syntax

Pascal

```
p_shmdt (shmadr);
```

FORTRAN

```
SHMDT (SHMADR)
```

Parameters

shmadr
is the address at which the memory segment is detached from the address space of the calling process. It is the same address as that at which the segment was originally attached (see **SHMAT** on page 10-2)

- In Pascal, *shmadr* is of type integer.
- In FORTRAN, *shmadr* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SHMDT** system routine, which in these examples detaches the shared-memory segment identified by the address returned by a call to **SHMAT**.

Pascal

```
procedure shmdt1;

const
  %include/usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, grey, pink, red : integer;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  brown := 'm';
  orange := '/tmp/junk';
  grey := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  pink := p_shmget (red, 512, grey);
  blue := p_shmat (pink, 0, 0);
  green := p_shmdt (blue);
  writeln (green)
end;
```

FORTRAN

```
SUBROUTINE SHMDT1
INCLUDE (/usr/include/ailfconsts.inc)
INTEGER SHMDT, SHMAT, SHMGET, FTOK
INTEGER BLUE, GREEN, RED, GREY, PINK
CHARACTER BROWN, ORANGE*80
BROWN = 'm'
ORANGE = '/tmp/junk '
GREY = IPCCRT + IRUSR
RED = FTOK (ORANGE, BROWN)
PINK = SHMGET (RED, 512, GREY)
BLUE = SHMAT (PINK, 0, 0)
GREEN = SHMDT (BLUE)
PRINT *, GREEN
END
```

SHMGET
— get a shared-memory-segment identifier

Description

The **SHMGET** system call returns a shared-memory-segment ID associated with the specified *key* value.

Syntax

Pascal

```
p_shmget (key, size, shmflg);
```

FORTRAN

```
SHMGET (KEY, SIZE, SHMFLG)
```

Parameters

key

is either the value 0 (IPCPVT) or an IPC key returned by the **ftok** system subroutine. A shared-memory ID, its associated data structure, and shared-memory segment, equal in bytes to the value of *size* is created if:

- *key* is set equal to **0** (IPCPVT).

  or

- *key* does not already have a shared-memory ID associated with it *and* the *shmflg* parameter is set equal to the constant IPCCRT.

The initial values of the data structure associated with a newly created shared-memory ID are listed later in this section under **Return Values**

- In Pascal, *key* is of type integer.
- In FORTRAN, *key* is of type INTEGER.

size
is the number of bytes in the shared-memory segment.

- In Pascal, *size* is of type integer.
- In FORTRAN *size* is of type INTEGER.

shmflg
specifies a set of conditions (options) governing the creation of a shared-memory data structure and the accessibility of the segment. The parameter value is that of one of the following options or is constructed from two or more of those options by logical ORing. The options are defined as constants in the Pascal and FORTRAN constants include files.

IPCCRT creates a data structure if one does not exist.

IPCEXL causes **SHMGET** to fail if IPCCRT is also set and the data structure already exists.

IRUSR permits the process that owns the data structure to read it.

IWUSR permits the process that owns the data structure to modify it.

IRGRP permits the group associated with the data structure to read it.

IWGRP permits the group associated with the data structure to modify it.

IROTH permits others to read the data structure.

IWOTH permits others to modify the data structure.

Return Values

A shared-memory ID is returned upon successful completion of the call. The data structure associated with a newly created ID (smds; see Appendix C) is initialized. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **SHMGET** system routine, which in these examples returns a shared-memory identifier associated with the value of *key* ("red") returned by the call to the **ftok** system subroutine.

Pascal

```
procedure shmget1;

const
  %include /usr/include/ailpconsts.inc
type
  %include /usr/include/ailtypes.inc
var
  blue, green, red : integer;
  orange : st80;
  brown : char;

%include /usr/include/aildefs.inc

begin
  brown := 'm';
  orange := '/tmp/junk';
  blue := IPCCRT + IRUSR;
  red := p_ftok (orange, brown);
  green := p_shmget (red, 512, blue);
  writeln (green)
end;
```

FORTRAN

```
      SUBROUTINE SHMGET1
      INCLUDE (/usr/include/ailfconsts.inc)
      INTEGER SHMGET, FTOK, BLUE, GREEN, RED
      CHARACTER BROWN, ORANGE*80
      BROWN = 'm'
      ORANGE = '/tmp/junk '
      BLUE = IPCCRT + IRUSR
      RED = FTOK (ORANGE, BROWN)
      GREEN = SHMGET (RED, 512, BLUE)
      PRINT *, GREEN
      END
```

Chapter 11. System Utilities

The following system calls invoke a variety of functions that can be generally, if loosely, described as operating system utilities.

Call	Function
CHDIR	changes the default directory.
REBOOT	restarts the operating system.
STIME	sets the system clock.
TIME	gets the system time.
UNAME, UNAMEX	get the name of the current operating system.

CHDIR
— change the default directory

Description

The **CHDIR** system call replaces the current working directory with the directory specified in the call. The current working directory is the starting point for searches when "/" is not specified.

Syntax

Pascal

```
p_chdir (path);
```

FORTRAN

```
CHDIR (PATH)
```

Parameters

path
is the name of the directory that becomes the current working directory when the call is issued. Assigning "dot dot" (..) to this variable specifies the parent of the current directory.

- In Pascal, *path* is a string variable or constant of type st80.
- In FORTRAN, *path* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

The value 0 (zero) is returned when the directory is changed. The value -1 is returned and an error code is set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **CHDIR** system routine. The directory specified in the call is /usr/games, which becomes the current working directory. The return value of the call is in the variable "folio". When the calling program terminates, the directory from which that program was executed once again becomes the current working directory.

Pascal

```
procedure chdir1;

type
  %include /usr/include/ailtypes.inc
var
  folio : integer;
  red : st80;

%include /usr/include/aildefs.inc

begin
  red := '/usr/games';
  folio := p_chdir (red);
  writeln (folio)
end;
```

FORTRAN

```
      SUBROUTINE CHDIR1
      INTEGER  FOLIO, CHDIR
      CHARACTER*80 RED
      RED = '/usr/games '
      FOLIO = CHDIR (RED)
      PRINT *, FOLIO
      END
```

REBOOT
— restart the operating system

Description

The **REBOOT** system call restarts the AIX Operating System in its normal default (non-maintenance) mode. The command can also specify a restart of the virtual-resource manager (VRM).

Syntax

Pascal

```
p_reboot (dev);
```

FORTRAN

```
REBOOT (DEV)
```

Parameters

dev
is the block special file from which the operating system is restarted. The value 0 (zero) restarts the system; a value of 1 restarts both the system and the VRM.

- In Pascal, *dev* is of type integer.
- In FORTRAN, *dev* is of type INTEGER.

Return Values

There is no return value from a successful **REBOOT** call. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **REBOOT** system routine. The return value of the call is in the variable "strider".

Pascal

```
procedure reboot1;

type
  %include /usr/include/ailtypes.inc
var
  dev, strider : integer;

%include /usr/include/aildefs.inc

begin
  dev := 0;
  strider := p_reboot (dev)
end;
```

FORTRAN

```
      SUBROUTINE REBOOT1
      INTEGER REBOOT, DEV, STRIDER
      DEV = 0
      STRIDER = REBOOT (DEV)
      END
```

STIME
— set the system clock

Description

The **STIME** system call sets the system's internal clock to a time and date that are calculated from a value specified in the call.

Note: Only the super-user may issue this call.

Syntax

Pascal

```
p_stime (tp);
```

FORTRAN

```
STIME (TP)
```

Parameters

tp
is the number of seconds that have elapsed since 00:00:00 January 1, 1970 GMT. Given this number, the routine calculates the time and date and resets the system's internal clock accordingly.

- In Pascal, *tp* is of type integer.
- In FORTRAN, *tp* is of type INTEGER.

Return Values

The value 0 (zero) is returned upon successful completion of the call. The value -1 is returned and an error code set in *errno* if the call is issued by anyone other than the super-user or if it fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **STIME** system routine. The value of "cronos" is the interval (in seconds) between 00:00:00 January 1, 1970 GMT and the time to which the system clock is to be set. The return value of the call is in the variable "titan".

Pascal

```
procedure stime1;

type
  %include /usr/include/ailtypes.inc
var
  cronos, titan : integer;

%include /usr/include/aildefs.inc

begin
  cronos := 31536000;
  titan  := p_stime (cronos);
  writeln (titan)
end;
```

FORTRAN

```
      SUBROUTINE STIME1
      INTEGER STIME, CRONOS, TITAN
      CRONOS = 31536000
      TITAN  = STIME (CRONOS)
      PRINT *, TITAN
      END
```

TIME
— get the system time

Description

The **TIME** system call returns the length of the interval (in seconds) from 00:00:00 Jan. 1, 1970 GMT to the current (system) time.

Syntax

Pascal

```
p_time (tloc);
```

FORTRAN

```
TIME (TLOC)
```

Parameters

tloc
is a variable that receives the length of the interval (in seconds from 00:00:00 Jan. 1, 1970 GMT to the current time) upon return from the call.

- In Pascal, *tloc* is of type integer.
- In FORTRAN, *tloc* is of type INTEGER.

Return Values

The current time is returned upon successful completion of the call. When the value returned is other than 0 (zero), it is also stored in the location to which *tloc* points.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here call the **TIME** system routine. The length of the interval, expressed in seconds, is returned in the variable "perdu". The return value of the call is in the variable "temps".

Pascal

```
procedure time1;

type
  %include /usr/include/ailtypes.inc
var
  temps, perdu : integer;

%include /usr/include/aildefs.inc

begin
  temps := p_time (perdu);
  writeln (perdu)
end;
```

FORTRAN

```
      SUBROUTINE TIME1
      INTEGER TIME, TEMPS, PERDU
      TEMPS = TIME (PERDU)
      PRINT *, PERDU
      END
```

UNAME, UNAMEX
— get the name of the current operating system

Description

The **UNAME** and **UNAMEX** system calls retrieve and store information that identifies the current operating system. They store this information in a data structure specified in the call.

The **UNAMEX** call is used in local area networks where a binary node is appropriate.

Syntax

Pascal

```
p_uname (name);

p_unamex (xname);
```

FORTRAN

```
FUNCTION UNAME (NAME)

FUNCTION UNAMEX (XNAME)
```

Parameters

name
is used only with the **UNAME** call. It points to the appropriate data structure (unam).

- In Pascal, *name* is of type unam.
- In FORTRAN, *name* is an array(5) of type CHARACTER*9.

xname
is used only with the **UNAMEX** call. It points to the appropriate data structure (xunam).

- In Pascal, *xname* is of type xunam.
- In FORTRAN, *xname* is an array(4) of type INTEGER.

Return Values

A nonnegative number is returned upon successful completion of the call (see **Notes**). The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown here print the name of the current operating system. The return value for **UNAME** is in the variable "nemo". Other information returned concerning the current operating system is located in the four remaining fields of the record "verne".

The **UNAMEX** call, which is used in a local-area-network environment, will return the binary node number in a variable parameter of type xunam.

Pascal

```
procedure uname1;

type
  %include /usr/include/ailtypes.inc
var
  verne : unam;
  nemo : integer;

%include /usr/include/aildefs.inc

begin
  nemo := p_uname (verne);
  writeln (verne.sysname)
end;
```

FORTRAN

```
      SUBROUTINE UNAME1
      INTEGER UNAME, NEMO
      CHARACTER*9 VERNE(5)
      NEMO = UNAME (VERNE)
      PRINT *, VERNE(1)
      END
```

Notes

If the unamx.nid field of the parameter's return value is a negative number, add 4 294 967 296 to that number to obtain the correct value.

Appendix A. Error Codes and Error Messages

This appendix describes the errors that can occur when a system call is issued. Some subroutines that invoke system calls indicate errors in a similar way.

System calls indicate the occurrence of an error by returning a special value. This value is almost always -1, but you should check the description of the particular system call to be sure. A number identifying the error condition is stored in an external variable called *errno* (see "Return Values, Error Codes, and Error Messages" in Chapter 1 for information on how to access *errno*). This variable is not cleared when a system call is successful, so its value is meaningful only after one error has occurred and before another.

The **errno.h** header file declares the *errno* variable and defines the name of each error condition.

For each error code the following list gives the code number, the symbolic name defined in the **errno.h**, header file, and the associated error message. (For additional information, see **perror** in *AIX Operating System Technical Reference.*)

1 EPERM Not the owner
2 ENOENT No such file or directory
3 ESRCH No such process
4 EINTR Interrupted system call
5 EIO I/O error
6 ENXIO No such device or address
7 E2BIG Argument list too long
8 ENOEXEC Exec format error
9 EBADF Bad file number
10 ECHILD No child process
11 EAGAIN No more processes
12 ENOMEM Not enough space
13 EACCES Permission denied
14 EFAULT Bad address
15 ENOTBLK Block device required

16 EBUSY Mount device busy
17 EEXIST File exists
18 EXDEV Cross-device link
19 ENODEV No such device
20 ENOTDIR Not a directory
21 EISDIR Is a directory
22 EINVAL Invalid argument
23 ENFILE File table overflow
24 EMFILE Too many open files
25 ENOTTY Not a typewriter
26 ETXTBSY Text file busy
27 EFBIG File too large
28 ENOSPC No space left on device
29 ESPIPE Illegal seek
30 EROFS Read-only file system
31 EMLINK Too many links
32 EPIPE Broken pipe
33 EDOM Math argument
34 ERANGE Result too large
35 ENOMSG No message of desired type
36 EIDRM Identifier removed
37 ECHRNG Channel number out of range
38 EL2NSYNC Level 2 not synchronized
39 EL3HLT Level 3 halted
40 EL3RST Level 3 reset
41 ELNRNG Link number out of range
42 EUNATCH Protocol driver not attached
43 ENOCSI No CSI structure available
44 EL2HLT Level 2 halted
45 EDEADLK Potential deadlock

Appendix B. Pascal and FORTRAN Constant Definitions

The following definitions of constants are required for Pascal and FORTRAN calling sequences.

OPEN constants

```
CREATE  = 256  (open file with file create
                used with the third argument of OPEN)
TRUNC   = 4096 (open file with truncation)
EXCL    = 8192 (exclusive open)
```

OPEN and CREAT constants

```
RDONLY  = 0 (read only)
WRONLY  = 1 (write only)
RDWR    = 2 (read and write)
NDELAY  = 4 (nonblocking I/O)
APPEND  = 8 (sets file pointer to end of file prior to writes)
```

OPEN, CREAT and CHMOD constants

```
IFMT    = 61440 (type of file)
IFDIR   = 16384 (directory)
IFCHR   = 8192  (character special)
IFBLK   = 24576 (block special)
IFREG   = 32768 (regular)
IFIFO   = 4096  (fifo)
ISUID   = 2048  (set user ID on execution)
ISGID   = 1024  (set group ID on execution)
ISVTX   = 512   (save text even after use)
IREAD   = 256   (read permission, owner)
IWRITE  = 128   (write permission, owner)
IEXEC   = 64    (execute/search permission , owner)
```

/usr/group definitions and MSGGET constants

```
IRWXU  = 448   (read, write, execute permission: owner)
IRUSR  = 256   (read permission: owner)
IWUSR  = 128   (write permission: owner)
IXUSR  = 64    (execute/search permission: owner)
IRWXG  = 56    (read, write, execute permission: group)
IRGRP  = 32    (read permission: group)
IWGRP  = 16    (write permission: owner)
IXGRP  = 8     (execute/search permission: group)
IRWXO  = 7     (read, write, execute permission: other)
IROTH  = 4     (read permission: other)
IWOTH  = 2     (write permission: other)
IXOTH  = 1     (execute/search permission: other)
ENFMT  = ISGID (enables enforcement-mode record locking)
```

Additional MSGGET constants

```
IPCCRT = 512   (create entry if key doesn't exist)
IPCEXL = 1024  (fail if key exists)
IPCALC = 32768 (use if identifier exists)
```

MSGRCV constants

```
IPCNER = 4096 (truncates a message that is too long)
IPCNWT = 2048 (specifies how to handle a non-existent message;
               also used in SEMOP as a sem_flg value)
```

Requst parameter of IOCTL

```
IOCTYP  = 65280 (returns the device type)
IOCINF  = 65281 (stores device information)
```

Device types

```
DDLP    = 'I'  (line printer)
DDTAPE  = 'M'  (mag tape)
DDTTY   = 'T'  (terminal)
DDDISK  = 'R'  (disk)
DDRTC   = 'c'  (real-time (calendar) clock)

DDPSEU  = 'Z'  (pseudo device)
DDNET   = 'N'  (networks)
DDEN    = 'E'  (Ethernet interface)
DDEM78  = 'e'  (3278/79 emulator)
```

Tape-drive types

```
STREAM  = 1 (streaming tape drive)
STRSTP  = 2 (start-stop tape drive)
```

Flags

```
DFIXED  = 01 (nonremovable)
DFRAND  = 02 (random access possible)
DFFAST  = 04 (a relative term)
```

GETGRP constants

```
NGROUP = 65536 (largest number of group access entries allowed)
```

USRINFO constants

```
INFSIZ  = 64 (size of user buffer)
GETINF  = 2
SETINF  = 1
```

Signal constants

```
SIGHUP = 1  (hangup)
SIGINT = 2  (interrupt (rubout))
SIGQIT = 3  (quit (ASCII FS))
SIGILL = 4  (illegal instruction (not reset when caught))
SIGTRP = 5  (trace trap (not reset when caught))
SIGIOT = 6  (IOT instruction (abort))
SIGDGR = 7  ((sigdanger) system is likely to crash soon)
SIGFPE = 8  (floating-point exception)
SIGKIL = 9  (kill (cannot be caught or ignored))
SIGBUS = 10 (bus error)
SIGSGV = 11 (segmentation violation)
SIGSYS = 12 (bad argument to system call)
SIGPIP = 13 (write on a pipe with no one to read it)
SIGALM = 14 (alarm clock)
SIGTRM = 15 (software termination signal from kill)
SIGU1  = 16 (user defined signal 1)
SIGU2  = 17 (user defined signal 2)
SIGCLD = 18 (death of a child)
```

```
SIGPWR = 19 (power-fail restart)
SIGAIO = 25 (basic LAN signal for asynchronous I/O)
SIGPTY = 26 (PTY device driver read/write availability)
SIGIO  = 27 (I/O intervention required)
SIGRNT = 28 (monitor mode grant)
SIGRTC = 29 (monitor mode retracted)
SIGSND = 30 (sound ack)
SIGMSG = 31 (data pending)

SIGDFL = 0  (for signal code parameter default)
SIGIGN = 1  (for signal code parameter ignore)
SIGFNC = 2  (for signal code parameter function address)
SIGADDR = 2 (for sigvec code parameter handler address)
```

SHMAT constants

```
SHMLBA = 268435456 (next lower segment boundary)
SHMRND = 8192      (round address to next lower segment boundary)
SHMMAP = 2048      (maps a file)
SHMRDO = 4096      (specifies read only mode)
SHMCPY = 16384     (maps a file in copy-on-write mode)
```

SEMCTL and MSGCTL constants

```
IPCRMD = 0 (remove identifier)
IPCSET = 1 (set options)
IPCSTT = 2 (get options)
GTNCNT = 3 (get semcnt value)
GETPID = 4 (get sempid value)
GETVAL = 5 (get semval value)
GETALL = 6 (get every semval value)
GTZCNT = 7 (get semzcnt value)
SETVAL = 8 (set semval value)
SETALL = 9 (set every semval value)
```

SEMOP constants

```
SEMNDO = 4096 (set up adjust on exit entry)
SEMODR = 8192 (perform operations non-automatically)
```

Appendix C. Pascal Type Declarations

The following declarations of types are required for Pascal calling sequences.

```
ushrt = 0..65535;
short = -32768..32767;
shrtptr = @short;
usign = integer;
cstring = packed array [1..81] of char;
cstrptr = @cstring;
cargv = array [1..80] of cstrptr;
st12 = string(12);
st12ptr = @st12;
cstr12 = packed array [1..13] of char;
st512 = string(512);
st512ptr = @st512;
st80 = string(80);
st80ptr = @st80;
pasargv = array [1..80] of st80;
charnine = packed array [1..9] of char;
char45 = packed array [1..45] of char;
piparray = array [1..2] of integer;
charptr = @char;
intptr = @integer;
```

Record used with Message, Semaphore, and Shared-Memory calls

```
perm = record
  uid : short;
  gid : short;
  cuid : short;
  cgid : short;
  mode : short;
  seq : short;
  key : integer
end;
```

Types associated with Message calls

```
msgxbuf = record
  mtime : integer;
  muid : short;
  mgid : short;
  mnid : short;
  mpid : short;
  mtype : short;
  mtext : st80
end;

msgxptr = @msgxbuf;
msgptr = @msg;

msg = record
  next : msgptr;
  mattr : msgxbuf;
  mtxtsz : short;
  mloc : short
end;

msgary = array [1..100] of msg;

msqid_ds = record
  msg_perm : perm;
  msg_first : msgptr;
  msg_last : msgptr;
  msg_cbytes : short;
  msg_qnum : short;
  msg_qbytes : short;
  msg_lspid : short;
  msg_lrpid : short;
  msg_stime : integer;
  msg_rtime : integer;
  msg_ctime : integer
end;

mdsptr = @msqid_ds;

msgbuf = record
  mtype : integer;
  mtext : st80
end;

mbufptr = @msgbuf;
```

Types associated with Shared-Memory calls

```
smds = record
  shperm : perm;
  shsegsz : integer;
  shsegid : short;
  shlpid : short;
  shcpid : short;
  shnattach : short;
  shcnattach : short;
  shatime : integer;
  shdtime : integer;
  shctime : integer
end;

smdsptr = @smds;
```

Types associated with Semaphore calls

```
sem = record
  semval : short;
  sempid : short;
  semcnt : short;
  semzcnt : short
end;

semptr = @sem;

semid_ds = record
  sem_perm : perm;
  sem_base : semptr;
  sem_nsems : short;
  semlcnt : short;
  sem_otime : integer;
  sem_ctime : integer
end;

semidptr = @semid_ds;
semary = array[1..1000] of short;
semaryptr = @semary;
```

```
abc = 0..2;
semrec = record
  case abc of
    0 : (val : integer);
    1 : (buf : semidptr);
    2 : (arry : semaryptr)
end;

sembuf = record
  sem_num : short;
  sem_op : short;
  sem_flg : short
end;

semopary = array [1..1000] of sembuf;
```

Types used in Signal calls

```
signalstack = record
  ss_sp : cstrptr;
  ss_onstack : integer
end;

stackptr = @signalstack;

signalvec = record
  sv_handler : intptr;
  sv_mask : integer;
  sv_onstack : integer
end;

sigvecptr = @signalvec;
```

Record used in the TIMES call

```
tms = record
  tms_utime : integer;
  tms_stime : integer;
  tms_cutime : integer;
  tms_cstime : integer
end;
```

Record used in the UTIME call

```
utimbuf = record
  actime : integer;
  modtime : integer
end;

utimptr = @utimbuf;
```

Record used in the UNAME call

```
unam = record
  sysname : charnine;
  nodename : charnine;
  release : charnine;
  version : charnine;
  machine : charnine
end;

unptr = @unam;

xunam = record
  nid : usign;
  reserved : array [1..3] of integer
end;

xunptr = @xunam;
```

Types used in the STAT and FSTAT calls

```
statrec = record
  st_dev  : integer;
  st_ino : short;
  st_mode : short;
  st_nlink : short;
  st_uid : short;
```

```
  st_gid : short;
  st_rdev : integer;
  st_size : integer;
  st_atime : integer;
  st_mtime : integer;
  st_ctime : integer
end;

statptr = @statrec;
```

Types used in the USTAT call

```
ustatrec = record
  f_tfree : integer;
  f_tinode : usign;
  f_fname : array [1..6] of char;
  f_fpack : array [1..6] of char
end;

ustatptr = @ustatrec;

devkind = (disk, map, ether, mag);
devinfo = record
  devtyp_flg : packed array [1..2] of char;
  hold : short;

  case devkind of
    disk :
        (bytpsec : short;          {bytes per sector}
         secptrk : short;          {sectors per track}
         trkpcyl : short;          {tracks per cylinder}
         numblks : integer);       {blocks this partition}

    map :
        (capab : char;             {capabilities}
         mode : char;              {current mode}
         hres : short;             {horizontal resolution}
         vres : short);            {vertical resolution}
```

```
    ether :
        (capabs : short;          {capabilities}
         haddr : array [1..6] of char);   {hardware
                                                address}
    mag :
        (typ : short)             {what flavor of tape}
                                  {defined in pconsts}
end;

devptr = @devinfo;


devinf = record
  devtype : char;
  flags : char;
  bytpsec : short;                {bytes per sector}
  secptr : short;                 {sectors per track}
  trkpcyl : short;                {tracks per cylinder}
  numblks : longint               {blocks this partition}
end;
```

Types used with the SIGVEC call

```
FP_STATUS = integer;
                    {HOLDS THE FOLLOWING INFORMATION:
                    bit 1 : SIGFPE on exception
                    bit 2 : exception occurred
                    bit 3 : invalid operation occurred
                    bit 4 : exception on invalid operation
                    bit 5 : divide by zero
                    bit 6 : exception on divide by zero
                    bit 7 : overflow occurred
                    bit 8 : exception on overflow
                    bit 9 : underflow occurred
                    bit 10 : exception on underflow
                    bits 11-21 : reserved
                    bits 22 & 23 : compare results
```

```
                    bits 24 & 25 : rounding mode
                    bit 26 : inexact results occurred
                    bit 27 : exception on inexact results
                    bits 28 & 29 : reserved
                    bits 30-32 : machine communications
                                 type}

choice = 0..2;
fpreg = record

case choice of
  0 : (hp : usign;
       lp : usign);
  1 : (dd : double);
  2 : (freg : array [1..2] of real)
end;

fptrapinfo = integer;

fptrap = record
  info : fptrapinfo;
  designated_result : fpreg
end;

fpvmach = record
  fpregarray : array [1..8] of fpreg;
  statusreg : FP_STATUS;
  fptrapvar : fptrap
end;

sigcontext = record
  sc_onstack : integer;      {Sigstack state to restore}
  sc_mask : integer;         {Signal mask to restore}
  sc_sp : integer;           {sp to restore (ignored)}
  sc_pc : integer;           {pc to restore}
  sc_ps : integer;           {psl to restore (ignored)}
  fpvmp : @fpvmach       {pointer to virtual fp machine}
end;

contexptr = @sigcontext;
```

Appendix D. Pascal Procedure and Function Declarations

The following declarations of procedures and functions are required for Pascal calling sequences.

```
procedure p_ercode; external;

procedure p_perror; external;


function p_access (path : st80; amode : integer) : integer;
                                                    external;

function p_acct (path : st80) : integer; external;

function p_alarm (sec: usign) : usign; external;

function p_brk (endds : integer) : integer; external;

function p_chdir (path : st80) : integer; external;

function p_chmod (path : st80; mode : integer) : integer; external;

function p_chown (path : st80; owner, group : integer) : integer;
                                                          external;

function p_chroot (path : st80) : integer; external;

function p_close (fildes : integer) : integer; external;

function p_creat (path : st80; mode : integer) : integer; external;

function p_dup (fildes : integer) : integer; external;

function p_eexit (status : integer ) : integer; external;
```

```
function p_execl (path,arg0,arg1,arg2,arg3 : st80) : integer;
                                                    external;

function p_execle (path,arg0,arg1,arg2,arg3 : st80;
                   envp : pasargv) : integer; external;

function p_execlp (filenm,arg0,arg1,arg2,arg3 : st80) : integer;
                                                     external;

function p_execv (path : st80; args : pasargv) : integer; external;

function p_execve (path : st80; args,envp : pasargv) : integer;
                                                      external;

function p_execvp (filenm : st80; args : pasargv) : integer;
                                                    external;

function p_exit (status : integer ) : integer; external;

function p_fclear (fildes : integer;nbytes : usign) : usign;
                                                      external;

function p_fcntl (fildes,cmd,arg : integer) : integer; external;

function p_fork : integer; external;

function p_fstat (fildes : integer; buf : statptr) : integer;
                                                     external;

function p_fsync (fildes : integer) : integer; external;

function p_ftok (path : st80; id : char) : integer; external;

function p_ftrunc (fildes : integer; len : usign) : integer;
                                                    external;

function p_getgid : integer; external;

function p_getpid : integer; external;

function p_getuid : integer; external;

function p_gtegid : integer; external;

function p_gteuid : integer; external;
```

```
function p_gtpgrp : integer; external;

function p_gtppid : integer; external;

function p_ioctl (fildes,requst : integer;
                  arg : devptr) : integer; external;

function p_kill (pid,sig : integer) : integer; external;

function p_link (path1,path2 : st80) : integer; external;

function p_lockf (fildes,requst,size : integer) : integer;
                                                  external;

function p_lseek (fildes,offset,whence : integer) : integer;
                                                    external;

function p_mknod (path : st80; mode,dev : integer): integer;
                                                    external;

function p_mount (dev,dir : st80; rwflag : integer) : integer;
                                                      external;

function p_msgctl (msqid,cmd : integer; buf : mdsptr): integer;
                                                       external;

function p_msgget (key,msgflg : integer) : integer; external;

function p_msgrv (msqid : integer; msgp : mbufptr;
                  msgsz,msgtyp,msgflg : integer) : integer; external;

function p_msgsnd (msqid : integer; msgp : mbufptr;
                  msgsz,msgtyp,msgflg : integer) : integer; external;

function p_msgxrv (msqid : integer; msgpt : msgxptr;
                  msgsz,msgtyp,msgflg : integer) : integer; external;

function p_nice (incr : integer) : integer; external;

function p_open (path : st80; oflag,mode : integer): integer;
                                                     external;

function p_pause : integer; external;
```

```
function p_pipe (fildes : piparray) : integer; external;

function p_plock (op : integer) : integer; external;

function p_profil (buf : shrtptr;
                   bufsiz,offset,scale : usign) : integer; external;

function p_ptrace (requst,pid,addr,data,buff : integer) : integer;
                                                          external;

function p_reboot (dev : integer) : integer; external;

function p_sbrk (incr : integer) : integer; external;

function p_semctl (semid : semidptr; semnum,cmd : integer;
                   var arg : semrec) : integer; external;

function p_semget (key,nsems,semflg : integer) : integer; external;

function p_semop (semid : integer; var sops ; semopary;
                  nsops : integer) : integer; external;

function p_setgid (gid : integer) : integer; external;

function p_setuid (uid : integer) : integer; external;

function p_shmat (shmid,shmadr shmflg : integer) : integer;
                                                    external;

function p_shmctl (shmid,cmd : integer; buf : smdsptr) : integer;
                                                          external;

function p_shmdt (shmadr : integer) : integer; external;

function p_shmget (key,size,shmflg : integer) : integer; external;

function p_sigblk (mask : integer): integer; external;

function p_signal (sig,action : integer; func : intptr): integer;
                                                          external;

function p_sigpas (sigmask : integer): integer; external;
```

```
function p_sigmsk (prvmask : integer): integer; external;

function p_sigstk (instack,outstack : stackptr) : integer;
                                                  external;

function p_sigvec (sig,code : integer;
                   invec,outvec : sigvecptr) : integer; external;

function p_stat (path : st80; buf : statptr) : integer; external;

function p_stime (tp : integer) : integer; external;

function p_stpgrp : integer; external;

function p_sync : integer; external;

function p_time (var tloc : integer) : integer; external;

function p_times (buf : tms) : integer; external;

function p_ulimit (cmd,newlim : integer) : integer; external;

function p_umask (cmask : integer) : integer; external;

function p_umount (dev : st80) : integer; external;

function p_uname (name : unam) : integer; external;

function p_unlink (path : st80) : integer; external;

function p_ustat (dev : integer; buf : ustatptr) : integer;
                                                   external;

function p_utime (path : st80; times : utimptr) : integer; external;

function p_wait (status : integer) : integer; external;

function p_unamex (xname : xunam) : integer; external;
```

Appendix E. The ftok System Subroutine

ftok
— generate an interprocess-communication key

Description

The **ftok** system subroutine returns a *key* that can be used to obtain interprocess-communication identifiers.

Syntax

Pascal

```
p_ftok (path, id);
```

FORTRAN

```
FTOK (PATH, ID)
```

Parameters

path

is the path name of an existing file that can be accessed by the calling process.

- In Pascal, *path* is of type st80.
- In FORTRAN, *path* is a string or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

id

is a character that uniquely identifies a project.

- In Pascal, *id* is of type char.
- In FORTRAN, *id* is of type CHARACTER.

Return Values

A *key* is returned upon successful completion of a call to **ftok**. The value -1 is returned and an error code set in *errno* if the call fails.

- In Pascal, the return value is of type integer.
- In FORTRAN, the return value is of type INTEGER.

Examples

The Pascal procedure and FORTRAN subroutine shown on the opposite page issue a call to the **ftok** system subroutine, which returns a key associated with the file /tmp/sample.

Pascal

```
procedure ftok1;

type
  %include /usr/include/ailtypes.inc
var
  red : integer;
  blue : st80;
  green : char;

%include /usr/include/aildefs.inc

begin
  green := 'z';
  blue := '/tmp/sample';
  red := p_ftok (blue, green);
  writeln (red)
end;
```

FORTRAN

```
      SUBROUTINE FTOK1
      INTEGER RED
      CHARACTER BLUE*80, GREEN
      GREEN = 'z'
      BLUE = '/tmp/sample '
      RED = FTOK (BLUE, GREEN)
      PRINT *, RED
      END
```

Appendix F. The perror System Subroutine

perror
— write a message

Description

The **perror** system subroutine writes a message explaining a system-call error.

Syntax

Pascal

```
p_perror (pmsg);
```

FORTRAN

```
PERROR (PMSG)
```

Parameters

pmsg
is a user-defined message that precedes the standard error message.

- In Pascal, *pmsg* is of type st80.
- In FORTRAN, *pmsg* is a string variable or constant of type CHARACTER*80. The terminating character of the string must be a blank space.

Return Values

There is no return value from a successful **perror** call.

Examples

The Pascal procedure and FORTRAN subroutine shown here will print an error code number and the associated error message if the *path* parameter (in the **CHDIR** call) specifies a non-existent directory.

Pascal

```
procedure showerror;

var
  result, code : integer;
  pmsg : st80;

begin
  pmsg = 'MEANING OF ERROR';
  result := p_chdir ('/usr/nonexist');
    if result = -1 then
      begin
        code := p_ercode;
        writeln (code);
        p_perror (pmsg)
      end
end;
```

FORTRAN

```
      SUBROUTINE ERRORS
      INTEGER RESULT, CODE
      RESULT = CHDIR ('/usr/nonexist ')
      IF (RESULT .EQ. -1) THEN
      CODE = ERCODE ()
      PRINT *, CODE
      CALL PERROR
      ENDIF
      END
```

Index

D

E

F

G

I

N

O

P

R

S

T

U

V

W

Z

Reader's Comment Form

IBM RT PC VS Language/Operating System Reference Manual SH23-0131-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

Reader's Comment Form

IBM RT PC VS Language/Operating System Reference Manual SH23-0131-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

International Business Machines Corporation
Department 79L, Building 4
Commerce Park and Eagle Road
Danbury, CT 06810

Printed in the
United States of America

SH23-0131